12-1-1997

Device independent color imaging modules for IRIS explorer

Christopher Hauf

Device Independent Color Imaging Modules for IRIS Explorer

Christopher R. Hauf

B.S. Imaging Science - Rochester Institute of Technology (1993)

A thesis submitted for partial fulfillment
of the requirements for the degree of
Master of Science in Color Science
in the Center for Imaging Science
of the College of Science
of the Rochester Institute of Technology

December 1997

Signature of Author

Roy Berns

Accepted by

College of Science
Rochester Institute of Technology
Rochester, New York

CERTIFICATE OF APPROVAL

M.S. DEGREE THESIS

The M.S. Degree Thesis of Christopher R. Hauf
has been examined and approved
by two members of the color science faculty
as satisfactory for the thesis requirement
for the Master of Science degree

Dr. Mark Fairchild, Thesis Advisor

Dr. Roy Berns

Thesis Release Permission Form

Rochester Institute of Technology
Center for Imaging Science

Title of Thesis Device Independent Color Imaging Modules for IRIS Explorer

I, _____, hereby grant permission to the Wallace Memorial Library of R.I.T. to reproduce this thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date 12/15/97

Device Independent Color Imaging Modules for IRIS Explorer

Christopher R. Hauf

Submitted for partial fulfillment
of the requirements for the degree of
Master of Science in Color Science
in the Center for Imaging Science
at the Rochester Institute of Technology

ABSTRACT

Having a flexible and expandable system of interconnecting modules which create device independent color image processing chains is a powerful tool for color science research and education. The objective of this research was to see if there existed a commercially available application which would provide the framework for such a system, learn that application, build a system of device independent imaging modules, and document the process so future developer's could easily expand the system. The Numerical Algorithms IRIS Explorer software proved to be the best fit for the system since it offered an intuitive user interface, a powerful set of default modules, and the capabilites for expansion through new module creation. With Explorer, a system of device independent color imaging modules were built, tested, and the module building process documented in this thesis meeting the objectives of the research.

Acknowledgements

I would like to thank the following people from the Munsell Color Science Lab for all of their help, assistance, and encouragement I received in completing this thesis research: Dr. Mark Fairchild for providing the concept for the system, for supporting me throughout the research , and for never giving up on me. Dr. Roy Berns for his comments and suggestions throughout the research. Lisa Reniff for her encouragement to keep going when things were not going my way. Finally, the students and visiting scholars of the Munsell Lab who shared their knowledge and gave their support to me throughout. Thanks.

Susan, thank you for your love, support, and encouragement. I could not have done it without you.

Meredith, thank you for your spirit, love, and support.

Mom and Dad, thanks for love and support over the years. You never stopped believing in me, and I will never stop believing in you. Simply put, thanks.

Table of Contents

List of Figures

List of Tables

1. Introduction

Having a flexible and expandable computer system with the ability to work with and interact with images in a device independent manner is a very powerful tool for color science research and teaching. This type of system would allow researchers to easily apply new algorithms to images, view the results, and determine the algorithms' effectiveness. At the same time, it would allow students of color to see how algorithms and images interact as algorithmic parameters are changed or different images are processed.

The Munsell Color Science Lab at the Rochester Institute of Technology, which acts as both a center for color science research and a center of color science education, would benefit greatly from such a system. With no system available commercially, the research outlined in this thesis was undertaken to see if such a core system could be built and documented using both existing technology within the Munsell Lab and new technologies available outside of the Munsell Lab.

The goal was to create a system that would be able to work on images in a device independent manner and would include color device models, color appearance models, device independent color space transformations, image processing routines, and 1D and 3D look-up table capabilities. The system needed to be flexible and easy to use with an intuitive user interface. It was required to be easily expandable, so that new algorithms beyond the scope of the research could be incorporated with a minimum of work. It needed to work with reasonable size images (512 pixels X 768 pixels) in near real time so that users could see how changes in the processing chain may effect their image, and it

needed to be able to process larger images efficiently for generating image databases for color and vision research.

To meet this goal would take a considerable amount of code generation just to create the framework for such a system leaving less time to work on the device independent color imaging aspects of the research. Therefore, a search was undertaken to see if there was a commercially available software package that would provide a usable, but expandable framework for working with images and that would be able to tap the processing power available on the Munsell Lab's Silicon Graphics UNIX workstations.

This thesis research outlines what was discovered, the reasons for the choices which were made, an overview of the final system that was built using the Numerical Algorithms Group's IRIS Explorer software, and documentation on how the modules were built.

2. Background

The following sections discuss the Numerical Algorithms Group's(NAG) IRIS Explorer software, an extensible, module based imaging and visualization tool, including why it was chosen, what IRIS Explorer is, and what different parts of Explorer were used in this research. This section provides the overview of Explorer necessary to understand how the modules were built for this research and how the entire system works as a reference for the reader for both the Experimental and Results and Discussion sections.

2.1 Why IRIS Explorer?

In order to build the system described in this thesis, a search was undertaken to find a commercially available software program which could provide the framework in which to build the system envisioned. Several programs are available that would fit this need. There was IRIS Explorer, Advanced Visualization Systems (AVS), and Khoros. At the time the search was undertaken, Khoros, a shareware, module based imaging program, was still in its formative stage, and AVS was quite capable, but required the purchase of the software & proper licenses. IRIS Explorer Version 2.2.2 came free with the Silicon Graphics IRIX 5.3 operating system software and seemed to have the capabilities needed for the creation of the system of interconnecting modules envisioned for this research. So, it was chosen as the candidate program.

At that time, IRIS Explorer was a product of Silicon Graphics Computer Systems, Inc. At Version 2.2.2, it came free with every copy of the SGI IRIX operating system. In 1994, SGI sold the rights to IRIS Explorer to the Numerical Algorithms Group(NAG), a non-profit organization out of the United Kingdom, which had been producing math and computation libraries for use in industry for many years. Explorer now requires a license and a licensing fee from NAG which differs whether the purchaser is an education institution or a business. At the time of this writing, NAG has had two version releases of IRIS Explorer since taking over the software and was in Version 3.5 for Silicon Graphics workstations. NAG also had ported the IRIS Explorer software to several other UNIX platforms(Sun Solaris, IBM AIX, HP-UX, Cray) and a port to the Windows NT platform.

Most of the development of this thesis was done under the 2.2.2 Version of Explorer, however, all of the software has been tested and run under the Version 3.5 release with very few problems which were mostly solved through the recompilation of the software using the IRIS Explorer 3.5 module building tools explained later.

2.2 What is IRIS Explorer and how does it work?

IRIS Explorer is a module based imaging and visualization tool. IRIS Explorer uses a module based paradigm to create a flexible tool for creating processing chains which are usable in many different applications from imaging to chemical structure visualization. In general, each Explorer module performs a single function on the data in the module. It is possible, however, for a module to perform more than a single operation on the data. Connected modules can be grouped within Explorer or modules can be written to perform multiple functions. In general, however, each module performs a single task such as reading an image into memory. This paradigm allows for the flexible connecting of modules to form, in the case of this research, device independent color image processing chains. It must be noted here, however, that the software created for the Explorer program can also function on patch type color data, however, its imaging application will be concentrated on in this document.

IRIS Explorer offers several execution modes. In the case of this thesis, the interactive mode will be discussed. It is also possible to run Explorer in other modes such as script mode where no user interface is presented and modules are executed through the use of script written in Explorer's scripting language, Skm. More information on Explorer's differing execution modes can be found in the IRIS Explorer

User's Guide[1] which is supplied with every copy of the IRIS Explorer software[1] or is available by navigating through NAG's website (http://www.nag.com/).

To use the IRIS Explorer modules and the module based processing paradigm in the interactive mode, the IRIS Explorer software is broken up into three different parts: the Map Editor, the Module Librarian, and the Log Window. When the Explorer software is run in interactive mode, all of these parts display on the screen allowing the user to begin working.

The Module Librarian is a customizable tool for the management of IRIS Explorer modules. Acting like a database, it provides a user interface where modules as well as IRIS Explorer maps, interconnected sets of Explorer modules, can be grouped for easy access. Fig. 1 shows a Module Librarian which has been customized for use in this research. Modules and maps for this research are displayed in the custom category called CRH_Thesis where modules are displayed in yellow and maps are displayed in blue. The Module Librarian also offers an additional feature called the module shelf. Noticeable at the bottom of Fig. 1, the module shelf is a location where commonly used modules may be placed for easy access. Fig. 1 shows modules for converting to and from the 1976 CIELAB color space to and from 1931 CIE XYZ normalized tristimulus values.

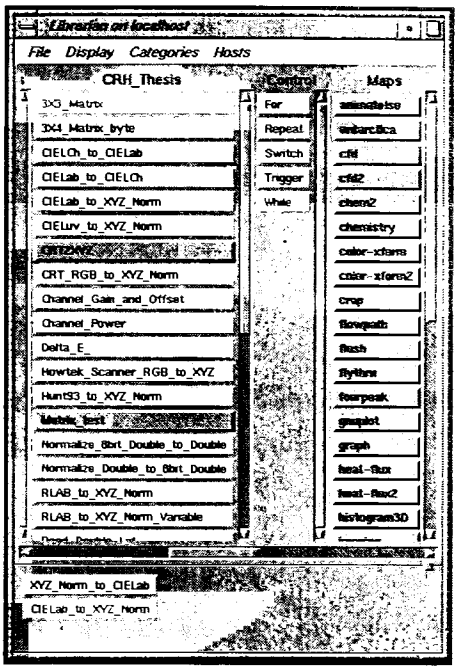


Fig. 1 Example setup for the IRIS Explorer Module librarian

Fig. 2 shows the Log Window which provides a window on the screen where data can be written to for many different uses including printing to the screen of different parameters inputted, outputted or calculated by a module and printing of different parameters when debugging new Explorer modules. The Log Window also provides a save function which allows the user to save to a text file all of the information currently buffered in the log window.

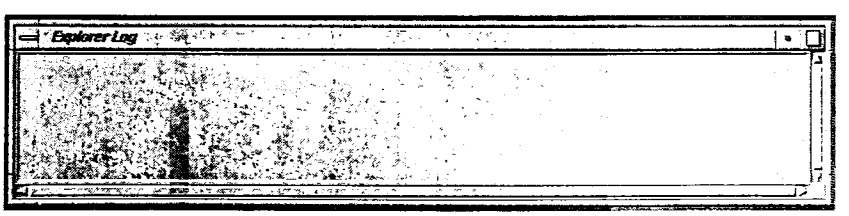


Fig. 2 The IRIS Explorer Log window

The Map Editor is an interactive canvas on which modules are placed and then wired together to form specific processing chains. Figure 3 shows the Map Editor window with a simple imaging chain. The chain includes three modules. The first module (starting from left to right) provides a listing of the directories and files on the system hard drive and allows the user to select a file. The middle module accepts the filename from the first module and reads the image into memory. The final module displays that image to the screen for viewing by the user.

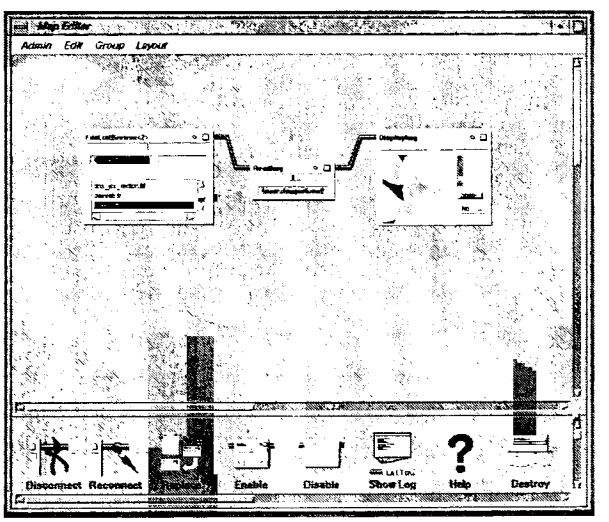


Fig. 3 NAG IRIS Explorer Map Editor window with sample Explorer map

When launched in the Map Editor, modules become visible as rectangular control panels. The size of the rectangle will depend upon the complexity of the module's user interface. Modules may use different types of widgets in their user interface. Widgets are different types of user interface controls such as slider bars, radio buttons, push buttons, text type in slots, image display windows, and etc. which allow for the input, control, and display of data in a module. The more widgets a module may have, the larger the

rectangle will be. However, a module can have no widgets at all if the processing algorithm in the module requires only the data entering on the input port(s).

Besides the differences due to the variation in need for widgets, all modules share some basic attributes including the module title or name, sizing buttons, input port(s) access pad, and the output port(s) access pad. These are shown in Fig. 4. The module title or module name is shown in the upper left hand corner. This matches the name in the Module Librarian. The sizing buttons allow the modules in the Map Editor to be further decreased in size or to be maximized for easier access to the user interface of a module if it has one.

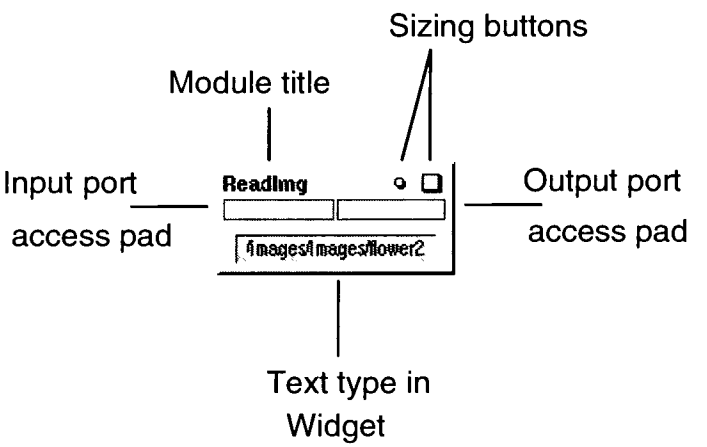


Fig. 4 Explanation of module control panel

To interconnect and share data, Explorer modules use the concept of ports. Input ports allow data to enter the module and output ports allow data to leave a module to be passed on to another module. The number of input and output ports is dependent on the function the module will perform. Explorer allows for multiple connections from a single output port to many different IRIS Explorer modules. The input and output ports

for a module are accessed through the port access pads shown in Fig. 4 which are also common to all modules.

Digging further into an Explorer module, the parts necessary to create a module can be decomposed into various distinct pieces. These parts provide different functionality at different levels of the Explorer module. Fig. 5 shows the various parts at the different levels. On the outside is the Module Control Wrapper(MCW) which is analogous to the main() function in a C program and oversees the overall control of the module. Internal to that is the Module Data Wrapper(MDW) which handles the conversions between incoming and outgoing data and the computational function shown at the center of Fig. 5. There is also the Module Generic Wrapper at the same level as the MDW. This wrapper contains some additional functionality common to all IRIS Explorer modules. The computational function, which is also referred to as the user function, is the location of the algorithms running inside of the Explorer module. By changing this function and adjusting the MDW and MCW, new IRIS Explorer modules can be built. The methods used to build the modules for this research are covered in both the Experimental section and the Results and Discussion section.

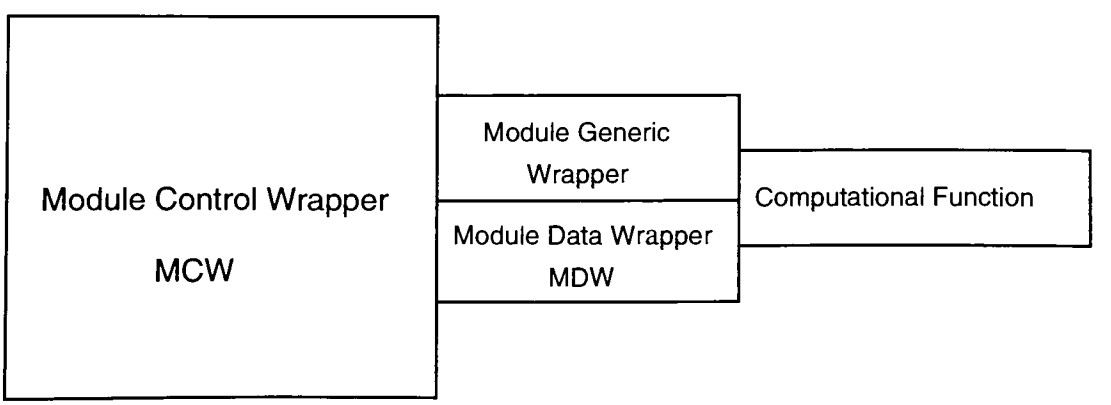


Fig. 5 Diagram of the various parts of an IRIS Explorer module

3. Experimental

The following sections discuss the details of the computing systems used, the software chosen as the system's framework, and the methods for the creation of the software for this system.

3.1 Computing systems

The Munsell Color Science Lab owns several Silicon Graphics(SGI) UNIX workstations. These workstations ranging from the low end INDY to the high end ONYX were all used in the creation of the software. Explorer, however, runs on all of the systems with the only difference being system performance. The major development system used was an SGI Indigo 2 XZ configured with a MIPS R4400 250MHz processor, 128 MB of RAM, 2 GB hard drive, Sony 20" monitor, and the SGI XZ graphics engine.

The Silicon Graphics workstations run a version of System V UNIX called IRIX. All development was done under IRIX Version 5.3. The current all system release of IRIX at the time of this writing was Version 6.2. All of the software was also run under IRIX 6.2, however, a recompilation of the source code was required to move the IRIX 5.3 binaries to IRIX 6.2 binaries. This was due to the use of different choices of the various libraries the development software used between the two releases of the SGI operating system and between two versions of the Explorer software used.

All software was written in ANSI C using the standard ANSI C compilers (cc) supplied with the SGI IRIX 5.3 and IRIX 6.2 releases.

3.2 Software

The software chosen to build the system on was Numerical Algorithms Group's IRIS Explorer. Three versions, V2.2.2, V3.0, V3.5, of Explorer were used during the development of the software for this thesis. The release version current with this writing for the Silicon Graphics workstations was Version 3.5.

Explorer Version 3.5 came standard with over 200 modules built in which were capable of many imaging and visualization operations including several modules tuned specifically for certain applications such as the visualization of chemical structures. Explorer also shipped with an additional suite of unsupported modules. Sourced from various places around the world, these modules have been released as shareware, but NAG provided no support for them. Most of the unsupported modules, however, were supplied with their source code. NAG also provided source code for approximately thirty-three percent of their supported modules which provided a good basis for learning about how to write IRIS Explorer modules.

Some of the standard modules proved useful for the system created which included file management, image file read and write, and image display. However, there was no support for device independent color imaging, but through the extensibility of Explorer, device independent color imaging capability was added through this research.

3.3 Explorer extension and module building

To extend Explorer, NAG supplied two additional programs, Module Builder and Data Scribe. Module Builder was used extensively in the creation of the modules for this

research. Data Scribe was used in a limited capacity, although it is a powerful tool for the creation of data input modules since Explorer has its own set of intrinsic data types which it uses to pass data between modules.

A full C Applications Programmers Interface (API) was also provided by NAG with Explorer. The API was used in conjunction with Module Builder to create the modules. In this research, the API was mainly used to manage the widgets used in many of the modules' control panels for the purpose of input/output of various types of module data. The API, however, was powerful enough to be used alone without the need for the Module Builder utility if a developer wanted total control of their code generation. In the case of this research, all of the algorithmic code was generated by the author using new code written or code already available in the Munsell Lab. The API was used on several occasions mainly for the control of widgets. However, the majority of the IRIS Explorer module interface code, the module data wrapper (MDW) and module control wrapper (MCW), was written by the Module Builder program. This was done to save time in code generation and debugging. Although not fully investigated, increased performance might have been one reason for writing all of the code. However, the level of performance achieved using Module Builder was quite sufficient for this research.

To create the modules for this research, the device independent color imaging algorithms were written into a standard ANSI C function, termed the user function or computational function in the Explorer literature. These C functions were based upon two of the five intrinsic data types which IRIS Explorer defines.

Since Explorer can be used for such a wide variety of data and uses the module based paradigm, it is critical that Explorer be able to have well-defined data structures for

the passing of different types of data from module to module. Explorer defines five data types as the basis for covering all types of input data. Explorer also provides the capabilities to create new intrinsic data types for Explorer. The five basic types of data structures are Parameters, Lattices, Pyramids, Geometries, and Picks.[2] When programming with these data types, a prefix *cx* is placed before the name of the data type. Also, the first letter of the data type is capitalized. For the Lattice data type, for example, it is referred to as *cxLattice* in the ANSI C code. Parameters define single pieces of data such as a value for the weight of a particular object. Lattices are multi-dimensional arrays of data. Images map directly into lattices since they are multi-dimensional arrays of pixels and pixel values. Pyramids are a hierarchy of lattices and carry connection information. They are used for the storage of data for rendering. Geometries store information about geometric shapes and are also used for rendering through the *Open Inventor* technology originally created by Silicon Graphics. Finally, the pick data type is used to store information taken from pointing and clicking on a certain location in a module display window.

This research only required the ability to pass single values and images, so only the parameter and lattice data types were used.

4. Results and Discussion

4.1 System design

Before starting the coding process, decisions on how the system should work and exchange data had to be made. These decisions included a list of algorithms required, a choice of a common interchange metric, a choice of data precision, a view of available

system resources, and many more. These decisions were necessary due to the module based paradigm of IRIS Explorer. In the traditional application paradigm, many of the same decisions would have to be made, but would all be hard coded to work together. In the case of this system, a flexible design had to be architected to allow modules to connect to each other and work as a system knowing that every module is independent of one another. Careful design was also important because Explorer already comes with many useful standard modules, and it was critical that the new modules generated would be able to work with already existing modules such as the ReadImg module or the DisplayImg module.

4.1.1 Algorithms to Modules

At the start, a list of possible algorithms to be supported was developed. The algorithms fell into several different categories including device models, color appearance models, color spaces, mathematical operations, look-up tables, data analysis, and data management. Table I represents the different algorithms from the device model, color appearance model, and color space categories which were supported in the final release of the software. Table II represents the different algorithms from the mathematical operations, look-up tables, data analysis and data management.

Table I. Device models, color appearance models, and color space conversions supported

Device models	Color appearance models	Color space support
Berns et al. CRT model	*RLAB*	*CIE 1931 XYZ*
Berns et al. scanner model	*Hunt*	*CIELAB*
Berns et al. film recorder model		*CIELCh*
		CIELUV
		sRGB
		Kodak PhotoYCC

Table II. Mathematical operations, look-up tables, data analysis and data management algorithms supported

Mathematical operations	Look-up tables	Data analysis	Data management
Sigmoid	*1D byte tables*	ΔE^*_{ab}	*Double to Byte*
Gain	*3D byte tables*		*Byte to Double*
Offset			*Double to Float*
Power			*Float to Double*
3X3 matrix multiply			

Many of these algorithms have both bi-directional transforms. For example, the Berns et al. CRT model has a transformation from CRT digital code values to colorimetry and an inverse transform from colorimetry to CRT digital code values. Therefore, any one algorithm might require at a minimum of two modules. To help understand how the actual modules would be built, a corresponding list of modules was then generated. Table III shows the device model algorithms and the corresponding modules built for each algorithm.

Table III. Device model algorithms versus modules built

Device algorithms	Modules
Berns et al. CRT model	CRT_to_XYZ_Norm
	XYZ_Norm_to_CRT
Berns et al. Scanner model	Howtek_Scanner_to_XYZ_Norm
Berns et al. Film Recorder model	Solitaire_to_XYZ_Norm
	XYZ_Norm_to_Solitaire

Most of these modules were built as originally envisioned, however, modifications were made based on other system decisions. In a few cases, for example, some of the functionality listed as separate modules was combined to form a single module. The *Channel_Gain_Offset* module is one example developed for this system. Since these operations are very often used in conjunction with one another in color

science, they were combined into a single module. The module still allows for independent control of the gain and offset on a per channel basis, but simplifies the system by providing a single module for this processing step.

The decision to combine functionality into a single module can both help and hinder a system such as the one outlined here. Providing multi-functional modules allows for decreased memory usage and fewer modules in an Explorer image processing chain. However, it also decreases the flexibility that a module based system delivers. In the traditional application paradigm, process A is hard coded to happen before process B which is hard coded to happen before process C, and so on. In imaging and color science research, it can be important to be able to test to see the results if the process ordering was varied such as C before B before A. In the module based paradigm Explorer provides, it is a simply a matter of disconnecting modules which do process A, B, and C and reconnecting in what whatever order a user desires. In a traditional application, the code would have to be modified to allow for this change in processing order.

The same is true for multi-functional modules. The processing order of the various functions is usually hardcoded into the module and would usually require a code modification to change the processing order. No code modification is required if the functionality is broken up into separate modules, but too many individual modules also have their problems. The solution is to try to strike a balance between combining operations while still maintaining system flexibility which was done for this system. The process of creating that balance, however, can depend on the specific system being modeled and may change the requirements of even the best balanced system.

4.1.2 Choice of interchange metric

For the system designed in this research, a decision had to be made as to what metric would be used to interchange data amongst the different modules to be built. Explorer provided the data structures needed to hold the data, parameters and lattices, but Explorer does not define what metric the numbers stored in those structures are in. The choice of a defined metric is important since the modules will need to interchange data. Since most of the algorithms used CIE 1931 2 degree XYZ tristimulus values[3] at some point, they were chosen as the interchange metric. One addition was also made. The XYZ tristimulus values used in this system are normalized to a given white point. This scales the values so that the white point is at 1.0 for each of the three tristimulus values.

This choice simplifies the calculations in some modules such as the conversions to and from CIELAB[3] which requires the choice and normalization by a given white point. However, it does complicate some other transformations which do not require normalized XYZ tristimulus values as input, but non-normalized tristimulus values as input. This complication manifests itself in two parts. The first is the need to keep track of the white point which the data was normalized to, and the second is an unnormalizing step in those modules which do not need normalized values on input. Overall, however, the level of complication is not very large. It does, however, take some understanding before a user can effectively and properly use the system.

4.1.3 Choice of precision

With the choice of normalized XYZ tristimulus values as the metric, the next required decision was at what precision would these numerical values be represented

inside the computer. This decision paralleled the requirement of looking at what the system resources would be available and at what level of performance would be acceptable for the system. The Silicon Graphics computer systems used in this research use a 64-bit architecture and have very high spec marks for floating point processing. Therefore, the decision was made that the processing time and performance would still be quite reasonable if the maximum precision was used given images in the neighborhood of 512 pixels by 768 pixels. So, the software written for this research uses ANSI C doubles which are equivalent to 64-bit floating point values on the SGI machines. This also removed any fear of doing calculations such as the normalization/unnormalization to the tristimulus values and losing accuracy due to round-off error. It is critical not to lose any precision for many research applications.

4.1.4 Performance

The actual performance of the completed system works as advertised and is reasonable given a 512 pixel by 768 pixel image. However, there are several caveats to this which were discovered during development of the system using such high precision values. IRIS Explorer uses a concept called shared memory to handle the memory requirements for a given set of modules. While all systems which were used in the development of this software have at least 64 megabytes of RAM, this shared memory area or arena in Explorer terminology usually resides on a hard drive on the system where Explorer is being run. The size and location of this shared memory arena can be set by the user through the use of the .explorerrc configuration file[1]. Without any user intervention, the default for Version 3.5 on the SGI is 150 megabytes of hard drive space.

As modules are opened in the Map Editor and are connected into a Explorer map, they start to use this shared memory arena to store and work on the data entering the module. The more modules opened and connected causes more of the shared memory area to be used until it is eventually exhausted. The reason for this stems from the module based paradigm in conjunction with a system design decision.

To help speed up processing as an Explorer map is being created, every module keeps a copy of any data which has been inputted into it in the shared memory arena even if that data has been passed on to any number of other modules through the output ports. The thinking is this. If a new module is added to the output port of a module which already has processed its data and has stored a copy in shared memory, the data is simply read out of memory and passed out the output port to the new module. Given that nothing has changed upstream of where the new module is added, none of the modules need to do any data processing. Data just gets read out of memory and passed on to the new module or modules. This concept works well if the data sets are small and where the maps may become very large or where a user wants to interactively add modules to a given map.

In the case of this research which is image centric, this system design choice combined with the maximum precision choice created some problems with large images or large maps consisting of many image processing modules. Most images used for this research were 8 bit per channel, video RGB images. For a 512 pixel by 768 pixel, this creates an image which requires 1.1 megabytes of storage. When that image is translated from its 8 bit representation to the 64 bit floating point representation used for this research, the image size is increased eight times. Therefore, the 1.1 megabyte image

becomes a 8.8 megabyte image, and every module which uses the 64 bit data representation requires the same 8.8 megabytes of the shared memory arena. This eight time multiplication is then increased by the number of modules which also use the 64 bit representation which can quickly use up the shared memory arena. This results in a failure of any module which oversteps the shared memory arena and prevents further processing.

There are several solutions to this problem and some were sought for this research. The first solution is to use a lower precision for the representation of the data in the modules. In the case of this research, floats, 32-bit floating point values, could have been substituted for doubles which would have halved the memory requirements. This change was not made to the code, but how it could be integrated was investigated. It is possible to write the user functions so that they will function for any given type of incoming data which the programmer chooses and not be restricted to a single representation. For this research, the change would allow users to choose between a double (64 bit) representation or a float (32 bit) representation depending upon their needs for accuracy versus processing capability. Future revisions of the software will probably incorporate this change to add flexibility.

Another solution is provided through the Explorer API and was also investigated, but not implemented in the final release of the software. Explorer uses a system called reference counting to keep track of what data is valid and should stay in shared memory and what data has become invalid and should be erased from the shared memory arena. The reference counts depend on the number of input and output ports connected. The Explorer software has been designed to do the reference counting by itself, so the user or

programmer of new Explorer modules does not need to be concerned with it. However, the Explorer API provides function calls to both increment and decrement the reference counts for a given module if a programmer chooses to supercede the default decisions made by the Explorer software.[4] It is possible to make a decrement call at the correct position in a user written computational function which effectively makes the module "forget" and free the part of the shared memory arena it has been using.

By making a module "forget" and free its space in the shared memory arena, the system requirements for memory are reduced to only those modules which are programmed not to "forget" or need to hold a copy of the image in memory such as the last module in a processing chain. The downside to this happens when new modules are added to the map. Since the module whose output port is being connected to no longer has a copy of its processed data in memory, it must seek new data on its input port and reprocess the data. Depending on how far down the processing chain the new module is being added and how many modules have been programmed to "forget", it will take additional time for new data to be processed through every module before getting to the newly added module. Therefore, by making modules "forget", system resources are freed, but processing time can be increased when interactively working with Explorer maps.

It should also be noted that this additional processing time penalty also occurs when a change is made to a widget in a given module. Just like adding a new module which requires data for processing, a change in a widget usually results in the need for a module to reprocess its data. To do so, it requests new data at its input port from the module it is connected to. If the upstream module has been designed to "forget", it too

will have to seek new data for reprocessing. The process continues up the chain until a module is found with data to process.

While showing a possible benefit in reducing the system demands for shared memory, the API calls for reference counting incrementing and decrementing were found to be difficult to manage when investigated as a quick solution. A better solution discovered was to use the API calls which flush a given output port of a module. This flushing appeared to provide the same result of making a module "forget", but did not require the additional complexity of managing the reference counting. This technique, however, was not implemented in the final system, but may be implemented in the future. Ideally, it would be advantageous to implement module "forgetting" as a choice on a per module basis. This would give the user the flexibility to decide for themselves which paradigm works best for their given processing task. Having modules "forget" is optimal for processing large images through a given imaging chain. Having modules hold their data is optimal for keeping the ability to work interactively on smaller images with an Explorer map where widget values may be changed or new modules added.

The final solution which was sought for this research was to use three dimensional look-up tables and three dimensional look-up table interpolation as an alternative image processing method for large images.[5] Instead of directly processing the images through the imaging chain, a 3D table of values would be processed through the imaging chain. These tables of equally spaced 8 bit digital code values range anywhere in size from 512 bytes to 60 kilobytes in size making them much smaller than even a 512 pixel by 768 pixel 8 bit per channel image which is 1.1 megabytes in size. The resulting processed 3D table of 8 bit code values can then be used in a conjunction with a three dimensional

look-up table interpolator and the input image to calculate the resulting pixel values for the output image. This concept is the foundation for the processing of images through the International Color Consortium (ICC) color management framework.[6] In the ICC case, both one dimensional and three dimensional tables can be used and stored in an ICC profile for the processing of color data through an ICC color management module or CMM.

Given the interpolator is written correctly and large enough tables are used, this method gives results very close to the processing of the image through the imaging chain of IRIS Explorer modules. However, since the 3D tables require significantly less storage space, processing efficiency is maximized while memory requirements are minimized for a given imaging chain. This method can be used for both interactive and batch processing of images. By connecting the interpolator to the end of an imaging chain, it is possible to interactively work with the modules in that chain where the initial 3D table gets reprocessed with any change of the module parameters, and the interpolator reprocesses the image at the end of the chain using the newly generated table. This method is also efficient for batch processing many images through the same imaging chain. 3D tables can be processed and stored for any number of given imaging chains. Those tables can then be used with the interpolator to process large numbers of images for a given experiment. This capability is critical to the Munsell Lab where large numbers of images are generated for psychophysical experimentation and had to be addressed in the system design along with allowing for the interactivity which is critical for teaching.

Overall, the decisions made for the overall design held through development. Different algorithms and support modules were added and subtracted from the final list as the development proceeded and needs changed. The up-front decision on an interchange metric allowed all modules to be developed to work together in the system. The realization of system limitations and the solutions sought were all viable. Each solution sought can play different roles depending on the final needs of the user for the system. One solution, three dimensional look-up tables and interpolation, was chosen since it provided the most flexibility, however, the other solutions may be incorporated at some level in a later refinement of the system.

4.2 Algorithms implemented

This section will briefly outline the algorithms that were implemented in this research. Most of these algorithms have been well documented in the color science literature, so readers of this document are encouraged to seek the proper references for more information on a given algorithm.

4.2.1 Device models

This section will highlight the different device models implemented for this research. Analytical modeling of color imaging devices is critical in color science research and education. On the research side, it allows for experimentation to take place without the need to be interactively working with a given device. This reduces the cost of materials which might be required to produce the desired experimental changes, reduces the time and requirements for color measurement, and increases the availability of a given device in a research laboratory environment. On the education side,

analytical modeling of color imaging devices gives students an intimate knowledge of how color is handled by different devices whether they are input devices analyzing color or output devices synthesizing color.

The software written for this research provides a selection of different color device models which have been developed outside of this research. Attempts have been made to allow the user the greatest amount of flexibility in managing the defined parameters for each model through the user interface of each device model module. As previously stated, for any given device model, there may be two modules at a minimum. One module which models the transformation from a device's device dependent color representation to a given device independent representation and the inverse which models the transformation from a device independent representation to a defined device dependent representation.

4.2.1.1 CRT Gain, Offset, Gamma model

The CRT or cathode ray tube is the primary color imaging device used to view and work with images in computers. The CRT or monitor acts as both an output device displaying images on the screen for a user to see, and an input device where the CRT acts as the canvas where a user creates an image or artwork or where the CRT acts as the decision maker as a user judges if adjustments to images are what they were anticipating or not. Therefore, it is critical to have an effective model of the CRT.

Developed by Berns, Motta, and Gorzynski[7], the CRT model implemented in this research uses an analytical equation to model the transformation from digital code values to colorimetry and the transformation from colorimetry back to digital code values for

display. Based on the three parameters, gain, offset and gamma, the model equation takes the form shown in Equation 1.

$$I = (gV + o)^{\gamma} \qquad (1)$$

where I is the intensity on the screen, g is the gain, o is the offset, V is the grid voltage, and γ is the non-linearity inherent in the electron gun in the CRT.

Through the luminance measurement of as few as five gray patches of varying luminance levels, and the use of non-linear optimization, it is possible to derive the gain, offset, and gamma parameters for the model equation for a given monitor in a given state. The colorimetric measurement of a white patch, a full drive red patch, a full drive green patch, and a full drive blue patch are also required to perform several tests on the additivity and stability of the CRT. These tests aid in understanding how well the model will fit the real measurements. The colorimetric measurements of the red, green, and blue patches are also required to derive a primary transformation matrix which relates the RGB tristimulus values for the CRT primaries to CIE XYZ tristimulus values.

The software enablement of this model for this research takes in the optimized model parameters and allows for the processing of data through the model. The parameters can be entered from the keyboard into the proper widget in the module. It is also possible to place the values in a specially formatted text file which the module can read. Fig. 6 shows a sample CRT data file, crttest.dat, included with the software.

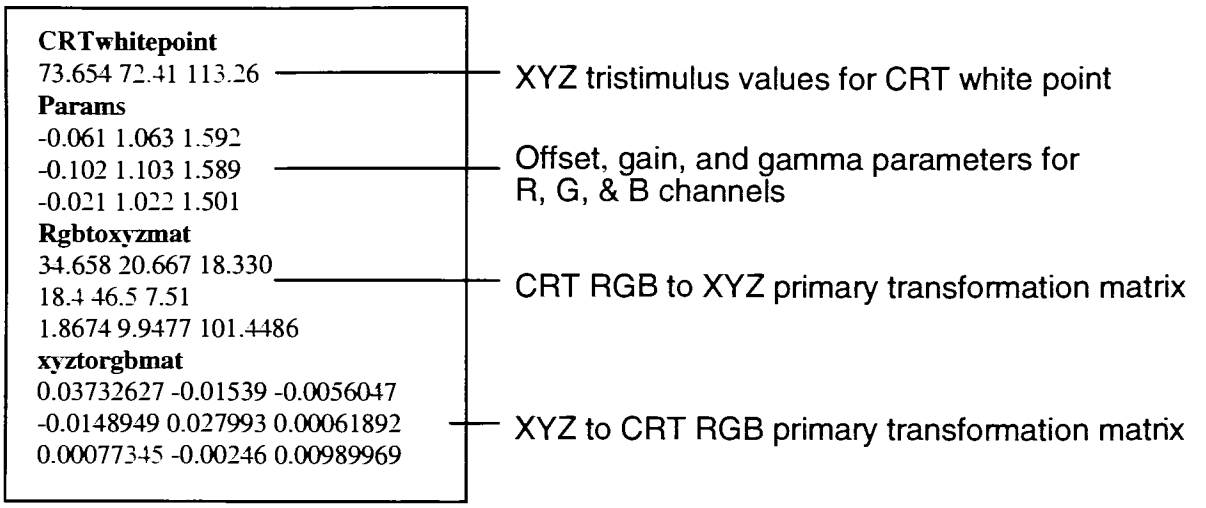


Fig. 6 Sample CRT module data file

The bolded text strings are comment lines and contain user readable information. They are ignored by the software. It should also be noted that a single CRT data file encapsulates the parameters for both the forward and inverse models with the inclusion of both the CRT RGB to XYZ matrix and the inverse of that matrix, XYZ to CRT RGB.

Both the forward model, colorimetry to digital code values, and the inverse model, code values to colorimetry, has been coded into separate modules for this research. In both cases, the digital code values are assumed to be 8 bit per channel, RGB code values and the colorimetry produced is normalized CIE XYZ tristimulus values based upon the primary transformation matrix derived from the CRT measurements and a normalizing white point inputted into the module. Fig. 7 shows the module control panel for the forward model module, *XYZ_Norm_to_CRT_RGB*.

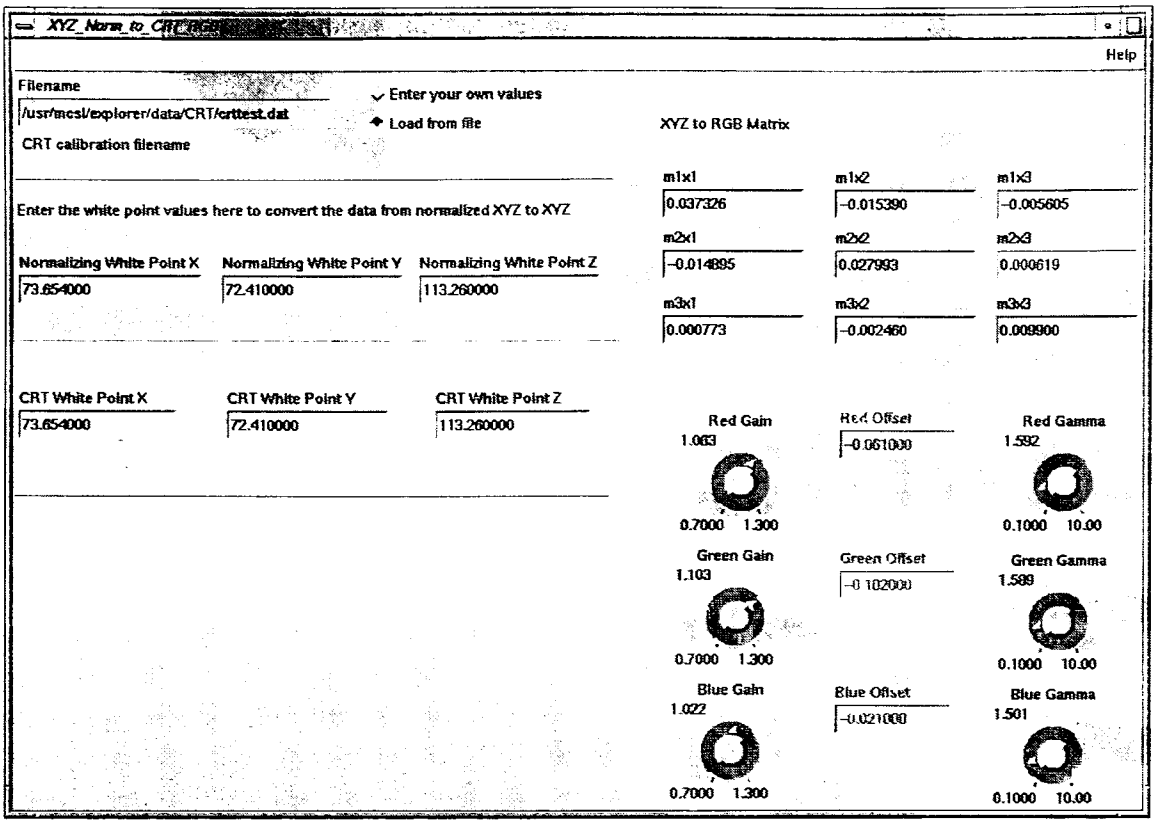


Fig. 7 Module control panel for module *XYZ_Norm_to_CRT_RGB*

This particular instance of this module shows CRT model data being read from a file which happens to be the same CRT data file shown in Fig. 6. The gain, offset, and gamma parameters have been read from the file and set. Since the gain and offset are constrained in the model to always equal one, the offset widgets are grayed out meaning the user can not change them directly. The offset value, however, is related to the gain control for each channel through the stated relationship, gain + offset = 1.0. Therefore, it is possible to change the offset by changing the gain. The primary transformation matrix has also been read and is displayed in the upper right hand corner of the module. The CRT white point has also been set as has the normalizing white point which was chosen for this instance of the module to be the same as the CRT white point. This normalizing

white point is used in this module to remove the normalization applied to the data prior to entering this module since the output of this module are digital code values.

Fig. 8 represents the inverse model encapsulated in the module named *CRT_RGB_to_XYZ_Norm*. The values are again derived for the sample CRT data file shown in Fig. X. Unlike the forward model module, there is no normalizing white point. The data is normalized to the CRT white point upon output.

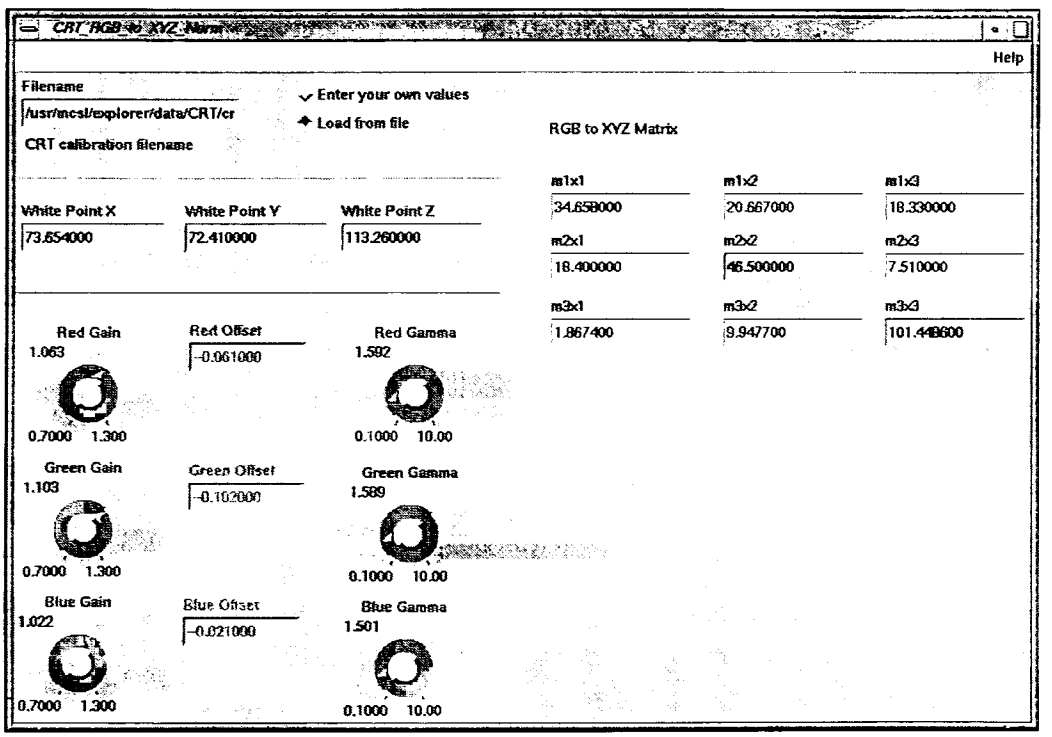


Fig. 8 Module control panel for module *CRT_RGB_to_XYZ_Norm*

4.2.1.2 Howtek scanner spectral model

A spectral scanner model was derived by Berns and Shyu[8]. This model was generated using a Howtek D4000 drum scanner which the Munsell Color Science Lab owns, but the approach to modeling can be expanded beyond just the Howtek scanner or

beyond the media types which were used in the original modeling and resulting scanner characterization.

Like the CRT model implemented in the software for this research, the Howtek scanner model and resulting module require a user to do all of their modeling prior to using this software which uses the derived model parameters to process image or patch data through the model. The module has been derived to work with model parameters derived from the Munsell Lab's Howtek scanner. Different media types are addressed through the input of different model data files beyond the defaults. The module should also work with other scanners modeled with this technique given the proper model input data files are changed. The default model data files, however, will be invalidated when a different scanner or media type is modeled.

Fig. 9 shows the control panel for the scanner model encapsulated in the module named *Howtek_Scanner_RGB_to_XYZ_Norm.* The module's control panel is very simple since it receives all of the model parameters by reading them from data files. The module has a default set of model input data files which are hardcoded into the module. These can be changed by modifying the source code and recompiling. The module, however, can also be set to "Enter your own" where a user can enter their own data files. In Fig. 9, the module has been set to use the default data files and their names and locations are shown in the five different text type in widgets. These data files include the spectral absorptivities for a given media type, the derived color matching functions (CMFS) for the scanner, a linearization matrix, a matrix relating code values to dye concentrations, and finally a file which provides information about the source used in the scanner.

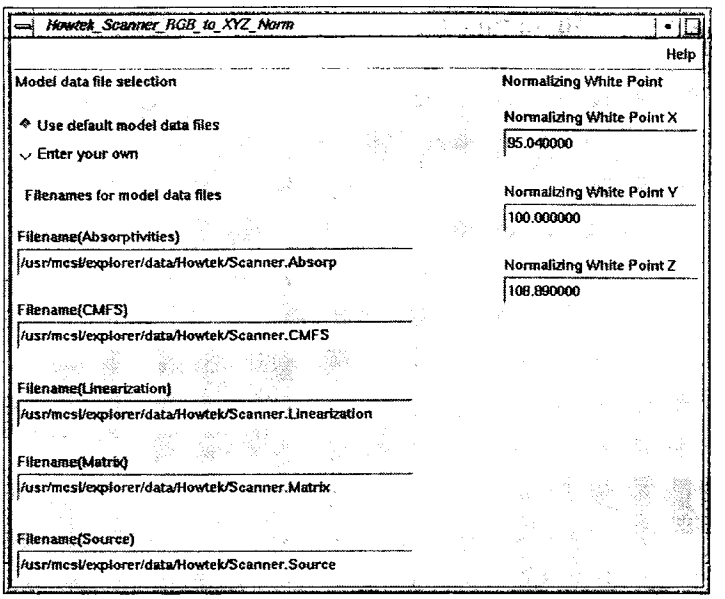


Fig. 9 Module control panel for module *Howtek_Scanner_RGB_to_XYZ_Norm*

Unlike the CRT model, this scanner model is uni-directional relating input scanner code values to output CIE colorimetry in the form of normalized XYZ tristimulus values. Shown in the upper right of Fig. 9, the module control panel provides a space to enter a normalizing white point which is set in this case to the 1931 XYZ tristimulus values for CIE illuminant D65.

4.2.1.3 Solitaire film recorder model

A spectral model for a Management Graphics Inc. Solitaire film recorder has been formulated by Fairchild et al.[9] Two IRIS Explorer modules have been written to apply this model. The first module relates normalized CIE XYZ values to the digital code values to be inputted into the device. The second module is the inverse model which relates the code values to normalized XYZ tristimulus values. Again, a user must

optimize their model parameters before using this module to process images through the model configured with the user's parameters.

Fig. 10 shows the module control panel for the *Solitaire_RGB_to_XYZ_Norm* module. The control panel for the *XYZ_Norm_to_Solitaire_RGB* module looks identical. The module has three text type in slots for the different required input data files. Default files are provided and can be selected through the radio button at the top of the module. Lastly, the module provides three text type in slots for the normalizing white point which is used in this case to normalize the output tristimulus values.

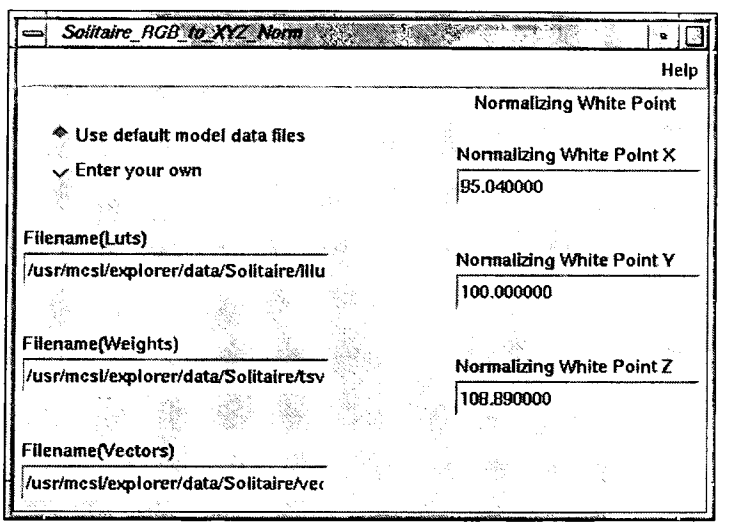


Fig. 10 Module control panel for the module *Solitaire_RGB_to_XYZ_Norm*

4.2.2 Color appearance models

Color appearance models are mathematical models which allow for the prediction of the appearance of color across disparate viewing conditions. Traditional colorimetry only allows for the prediction of color matches for a given, constant set of viewing conditions. In most applications of color imaging, however, there is a need to reproduce the appearance of color across different types of media and different sets of viewing

conditions. Therefore, a color appearance model is required to create the appropriate transformation, so that the appearance of a reproduced image closely resembles the appearance of the original image.

Many different models have been proposed, and at this writing, new models are under development. For this research, two color appearance models have been implemented as IRIS Explorer modules. They are the Hunt color appearance model and the RLAB color appearance model.

4.2.2.1 The Hunt color appearance model

Derived by Dr. R.W.G. Hunt, the Hunt color appearance model was first introduced in 1982. Since that time, it has been revised several times with the last revision in 1993 and published in 1994.[10] The Explorer modules created follow the 1993 revision. The Hunt model is complex requiring 20 different input parameters and the calculation of 42 different variables to derive the three output correlates of color appearance, hue (h), lightness (J), and chroma (C) as termed by Hunt.

The published equations for the Hunt model relate the colorimetry given in XYZ tristimulus values of a given color for a given set of viewing conditions to the set of Hunt color appearance correlates, JCh. For cross-media and cross-viewing condition modeling, however, an inverse model is also required which allows for the transformation of color appearance coordinates, JCh for the Hunt model, back to the colorimetry, XYZ, for a new set of viewing conditions. For the Hunt model, there is no set of equations to perform this inverse modeling. Therefore, an iterative procedure has been implemented in this research to provide an inverse model module. The module uses parameters

calculated from the forward model and a Newton-Raphson technique[5] to find the proper inverse.

Fig. 11 shows the control panel for the forward Hunt model module titled *XYZ_Norm_to_Hunt93.*

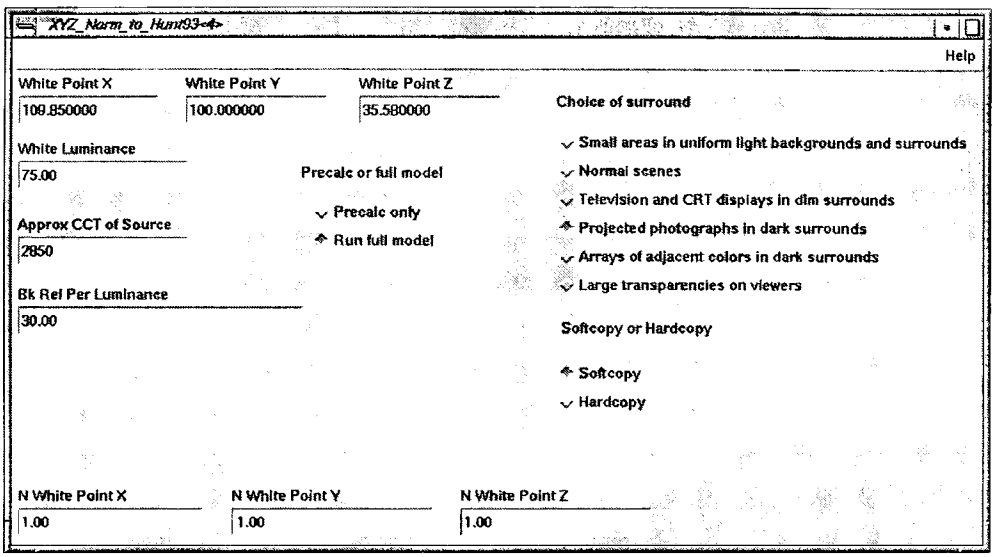


Fig. 11 Module control panel for the module *XYZ_Norm_to_Hunt93*

For this implementation of Hunt's model, the number of input parameters has been decreased to simplify the user interface. This simplification was possible since many of the input parameters can be pre-calculated from this smaller set of input values, and the module includes a switch which allows for only the input parameters to be pre-calculated. The ability to only pre-calculate input parameters is important when running the inverse model for a different set of viewing conditions. Since the inverse model depends on the forward model, a set of input parameters for the forward model used in the inverse calculation must be generated without running the full forward model. The module has been written to include an output port which outputs all of the pre-calculated Hunt parameters in the form of an Explorer lattice. The inverse model, shown in Fig. 12,

has a corresponding input port to accept the pre-calculated parameters from the forward model making the reverse model module control panel very simple with only widgets for a normalizing white point.

Overall the input parameters are self explanatory. The white point is required by Hunt to be the white point of the light source or illuminant to be used. The N white point is the normalizing white point used by this software to unnormalize the input normalized XYZ tristimulus values back to XYZ tristimulus values for calculation through the Hunt model.

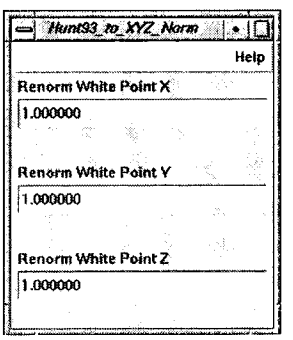


Fig. 12 Module control panel for the module *Hunt93_to_XYZ_Norm*

4.2.2.2 RLAB color appearance model

The second color appearance model implemented in this software is the RLAB color appearance model created by Fairchild and Berns.[11] RLAB has been designed to have color-appearance predictors similar to those of the CIELAB color space which are calculated through equations which are virtually identical to the equations used to calculate the CIELAB equations. However, the RLAB model provides a more accurate chromatic appearance transformation and allowance of variation of the power function nonlinearities, termed sigmas, in the CIELAB equations.

Fig. 13 shows the forward RLAB model module control panel which relates normalized XYZ tristimulus values to the RLAB coordinates, L^R, a^R, and b^R.

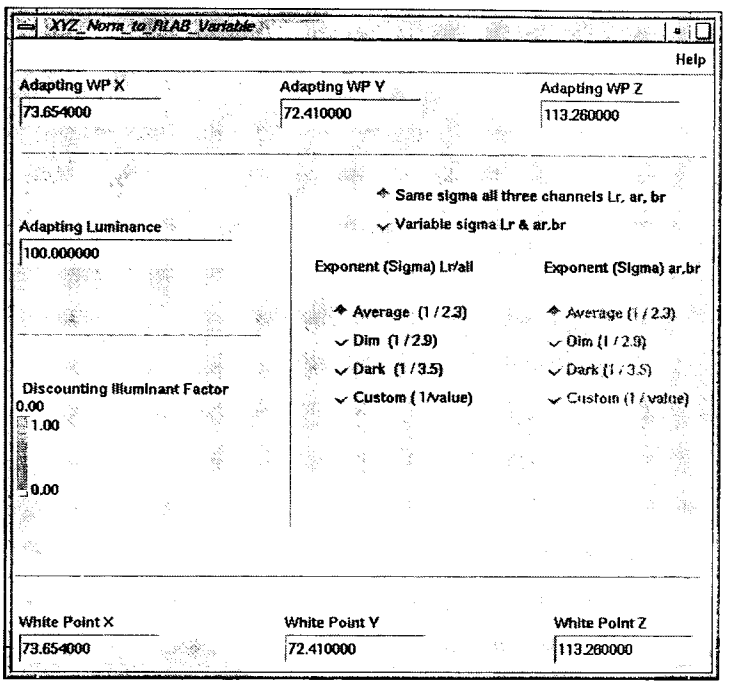


Fig. 13 Module control panel for the module *XYZ_Norm_to_RLAB_Variable*

The module control panel shown here is for the module titled *XYZ_Norm_to RLAB_Variable.* This module differs a little from the usual implementation of the RLAB model since it allows for different non-linearities to be applied to the lightness coordinate and the a^R and b^R coordinates together. Normally, the same non-linearity or sigma is applied to all three coordinates. This module is switchable to allow for both the normal mode of operation and for the variable mode of operation. There is also a module provided titled *XYZ_Norm_to_RLAB* which only allows for one sigma for all three coordinates.

Again, the white point shown at the bottom of the module is the normalizing white point which is used for the unnormalization of the input normalized XYZ tristimulus values since the RLAB model expects unnormalized values.

Unlike the Hunt model, the RLAB model has an inverse set of equations to relate the RLAB coordinates back to XYZ tristimulus values. Fig. 14 shows the module control panel for the module titled *RLAB_Variable_to_XYZ_Norm.*

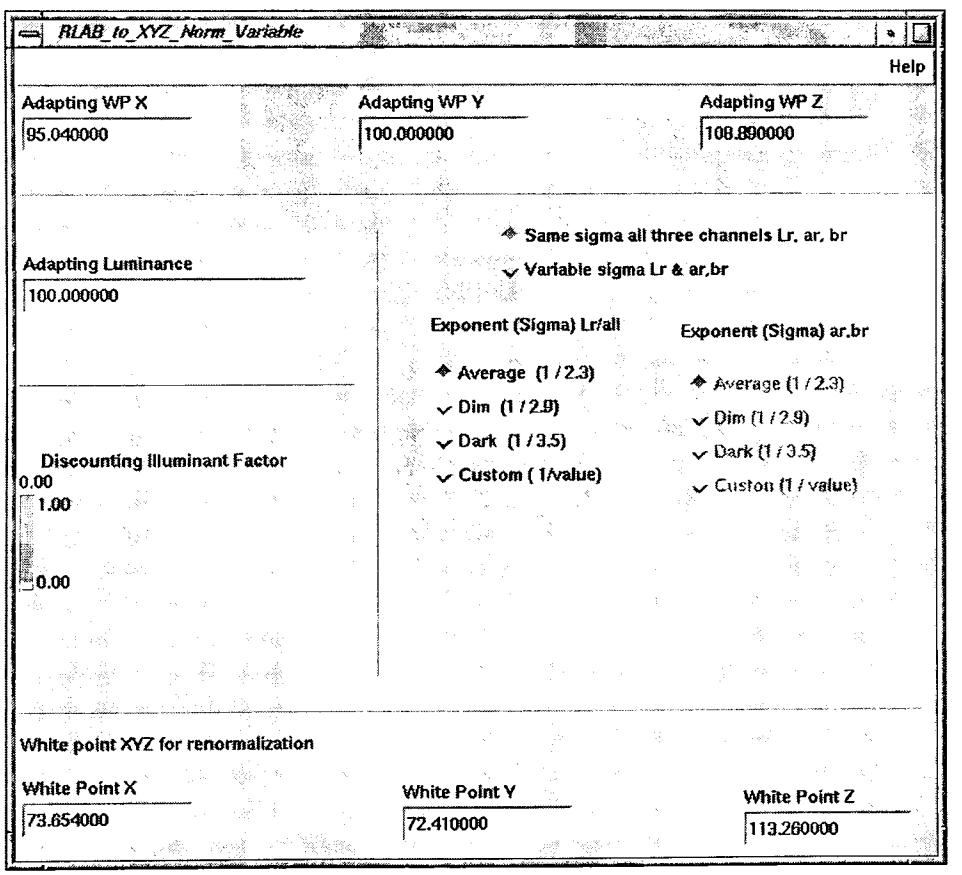


Fig. 14 Module control panel for module *RLAB_Variable_to_XYZ_Norm*

4.2.3 Color space transformations

To add additional flexibility to this system of modules, several different color space transformation modules have been written. Different applications require

different ways for color to be represented. Also, different types of color system analyses often require a specific color representation or are more effective for a given color representation. This is usually accomplished through the use of different defined color spaces.

For this research, modules were written to support the CIELAB, CIELCh, CIELUV, Kodak PhotoYCC, and sRGB color spaces.

4.2.3.1 CIELAB

In 1976, the CIE, Commission Internationale de l'Eclairage, standardized the CIELAB color space.[3] CIELAB was designed to be a visually uniform color space and provides predictors of lightness (L*) and chrominance (a*b*).

CIELAB is derived from CIE XYZ tristimulus values. The relationships are shown below in Equation 2.

$$
\begin{aligned}
L^* &= 116\left(\frac{Y}{Y_N}\right)^{1/3} - 16 \\
a^* &= 500\left[\left(\frac{X}{X_N}\right)^{1/3} - \left(\frac{Y}{Y_N}\right)^{1/3}\right] \\
b^* &= 200\left[\left(\frac{Y}{Y_N}\right)^{1/3} - \left(\frac{Z}{Z_N}\right)^{1/3}\right]
\end{aligned}
\qquad (2)
$$

where X_N, Y_N, and Z_N are the tristimulus values of a reference white and where $(X/X_N) > 0.008856$, $(Y/Y_N) > 0.008856$, and $(Z/Z_N) > 0.008856$. The CIE defines additional relationships for CIELab for ratios less than 0.008856, but they will not be described here. The software, however, does implement them.

For the CIELab modules, *XYZ_Norm_to_CIELab* and *CIELab_to_XYZ_Norm*, there are no widgets on the module control panels, therefore they are not shown in this document. A normalizing white point is not required since the CIELAB equations already use normalized tristimulus values defined by the X/X_N, Y/Y_N, and Z/Z_N relationships. Therefore, the reference white, X_N, Y_N, Z_N, is defined to be the normalizing white point of the normalized XYZ tristimulus values entering the *XYZ_Norm_to_CIELab* module or exiting the *CIELab_to_XYZ_Norm* module.

4.2.3.2 CIELAB CIELCh

At the same time that CIELAB was developed, the CIE also published a set of equations relating CIELAB coordinates for CIELCh coordinates where L^*_{ab} is the same lightness correlate used in CIELAB, C^*_{ab} is the correlate of chroma, and h_{ab} is the correlate of hue angle.[3] The CIELAB color space provides a Cartesian representation of color. CIELCh provides the same information in polar coordinate form.

Equation 3 defines the relationship for C^*_{ab} and for h_{ab} since L^* is the same as that given for CIELAB.

$$C^*_{ab} = \left[(a^*)^2 + (b^*)^2\right]^{1/2}$$
$$h_{ab} = \arctan\left(\frac{b^*}{a^*}\right) \qquad (3)$$

where h_{ab} defined in the following manner:

$0° <= h_{ab} < 90°$ if $a^* > 0, b^* >= 0$

$90° < h_{ab} <= 180°$ if $a^* < 0, b^* >= 0$

$180° <= h_{ab} <= 270°$ if $a^* < 0, b^* <= 0$

$270° <= h_{ab} <= 360°$ if $a^* > 0, b^* <= 0$

The module *CIELab_to_CIELCh* calculates the CIELCh values through these equations and the module *CIELCh_to_CIELab* calculates the inverse relationship which is also defined, but not shown here. Both of these modules have no module control panel widgets and require no white point information.

4.2.3.3 CIELUV

The CIE also defined in 1976 a second approximately uniform color space, CIELUV (L^*, u^*, v^*).[3] Based on the 1976 u', v' chromaticity coordinates which can be derived from XYZ tristimulus values, CIELUV was defined for the relationship shown in Equation 4.

$$
\begin{aligned}
L^* &= 116\left(\frac{Y}{Y_N}\right)^{1/3} - 16 \\
u^* &= 13L^*(u' - u'_n) \\
v^* &= 13L^*(v' - v'_n) \\
&\textit{where} \\
u' &= \frac{4X}{X + 15Y + 3Z} \\
v' &= \frac{9Y}{X + 15Y + 3Z}
\end{aligned}
\qquad (4)
$$

where u'_n and v'_n are the chromaticity coordinates of a reference white.

For this research, only the module relating the normalized XYZ tristimulus values to CIELUV coordinates was written. The module *XYZ_Norm_to_CIELuv* calculates CIELUV values. To do so, the module requires normalized XYZ tristimulus values and the normalizing white point. This is required due to the need to unnormalize the data for the calculation of the u', v' chromaticity coordinates required in the CIELUV calculation.

4.2.3.4 sRGB

Many desktop applications of color use 8 bit per channel, video RGB color spaces to encode colors for storage and usage on the desktop. The biggest issue, however, with video RGB on the desktop is the large number of different "video RGB's" which exist. In an attempt to remedy this situation, Stokes et al. have proposed a standard video RGB color space for use on the Internet called sRGB.[12] This color space has also been adopted for use in the *FlashPix*™ image file format[13], however, for Version 1.0 of the *FlashPix* format specification, it is referred to as NIFRGB. They are, however, the same color space.

sRGB is an 8 bit per channel RGB color space which has a defined reference monitor, defined set of digital encoding equations, and a defined reference viewing environment. An Explorer module was generated which moves sRGB code values to normalized XYZ tristimulus values. Equations 5 to 8 represent the transformation from the sRGB code values to colorimetry.

$$\begin{aligned} R'_{sRGB} &= R_{8bit} / 255.0 \\ G'_{sRGB} &= G_{8bit} / 255.0 \\ B'_{sRGB} &= B_{8bit} / 255.0 \end{aligned} \qquad (5)$$

For $R'_{sRGB}, G'_{sRGB}, B'_{sRGB} \leq 0.03928$

$$\begin{aligned} R_{sRGB} &= R'_{sRGB} / 12.92 \\ G_{sRGB} &= G'_{sRGB} / 12.92 \\ B_{sRGB} &= B'_{sRGB} / 12.92 \end{aligned} \qquad (6)$$

For $R'_{sRGB}, G'_{sRGB}, B'_{sRGB} \geq 0.03928$

$$R_{sRGB} = \left[\frac{(R'_{sRGB}+0.055)}{1.055}\right]^{2.40}$$

$$G_{sRGB} = \left[\frac{(G'_{sRGB}+0.055)}{1.055}\right]^{2.40} \qquad (7)$$

$$B_{sRGB} = \left[\frac{(B'_{sRGB}+0.055)}{1.055}\right]^{2.40}$$

The resulting R_{sRGB}, G_{sRGB}, and B_{sRGB} values are then rotated to XYZ tristimulus values through the matrix equation below.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix} \begin{bmatrix} R_{sRGB} \\ G_{sRGB} \\ B_{sRGB} \end{bmatrix} \qquad (8)$$

Finally, the XYZ tristimulus values are normalized by a white point inputted by the user, so that the values can be used with the other modules built for this system.

4.2.3.5 Kodak PhotoYCC Color Interchange Space

The Kodak PhotoYCC Color Interchange Space is a 8-bit per channel, luma/chroma color space which was originally used in the Kodak Photo CD system and has since been used in other Kodak products including the new *FlashPix* image file format.[13,14] Inherent in its design is the ability to encode a large color gamut, greater than the gamut of any video RGB, and encode a luminance dynamic range of up to 200% scene reflectance.

Since there are many digital images available in the PhotoYCC color space from Kodak Photo CD's, a module was built to transform the PhotoYCC code values to

normalized XYZ tristimulus values for use in the system created for this research. Equations 9 to 14 show the transformation

$$Luma = \left(\frac{1.402}{255}\right) * Y$$
$$Chroma_1 = \frac{C1 - 156}{111.40} \qquad (9)$$
$$Chroma_2 = \frac{C2 - 137}{135.64}$$

The resulting Luma and Chroma values are converted to ITU-R BT.709-2 nonlinear R'G'B'$_{709}$ values:

$$\begin{bmatrix} R'_{709} \\ G'_{709} \\ B'_{709} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & -0.194 & -0.509 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} Luma \\ Chroma_1 \\ Chroma_2 \end{bmatrix} \qquad (10)$$

The nonlinear R'G'B'$_{709}$ values are converted to linear RGB$_{709}$ values:

For $R'_{709}, G'_{709}, B'_{709} \geq 0.081$

$$R_{709} = \left(\frac{R'_{709} + 0.099}{1.099}\right)^{\frac{1}{0.45}}$$
$$G_{709} = \left(\frac{G'_{709} + 0.099}{1.099}\right)^{\frac{1}{0.45}} \qquad (11)$$
$$B_{709} = \left(\frac{B'_{709} + 0.099}{1.099}\right)^{\frac{1}{0.45}}$$

For $R'_{709}, G'_{709}, B'_{709} \leq -0.081$

$$R_{709} = -\left(\frac{R'_{709} - 0.099}{-1.099}\right)^{\frac{1}{0.45}}$$

$$G_{709} = -\left(\frac{G'_{709} - 0.099}{-1.099}\right)^{\frac{1}{0.45}} \qquad (12)$$

$$B_{709} = -\left(\frac{B'_{709} - 0.099}{-1.099}\right)^{\frac{1}{0.45}}$$

For $-0.081 < R'_{709}, G'_{709}, B'_{709} < 0.081$

$$R_{709} = \frac{R'_{709}}{4.5}$$

$$G_{709} = \frac{G'_{709}}{4.5} \qquad (13)$$

$$B_{709} = \frac{B'_{709}}{4.5}$$

The resulting linear RGB_{709} values are then rotation to CIE XYZ_{D65} values:

$$\begin{bmatrix} X_{D65} \\ Y_{D65} \\ Z_{D65} \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix} \begin{bmatrix} R_{709} \\ G_{709} \\ B_{709} \end{bmatrix} \qquad (14)$$

The XYZ_{D65} values are then normalized by a white point which is chosen by the user of the system.

4.2.4 Mathematical operations

In addition to device models, color appearance models, and color space transformations, a set of modules was created to perform some different mathematical operations to input data. These transformations operate pixel-wise, and are common to many different applications in color science. The operations include such operations as gain, offset, power, sigmoid, and 3x3 matrix multiplication.

For gain and offset, these operations were combined into a single module titled *Channel_Offset_Gain.* This module allows for channel independent gains and offsets to be applied to incoming data. The operation is applied to each individual channel through the relationship in Equation 15.

$$output = (gain * input) + offset \qquad (15)$$

For the power module titled *Channel_Power,* it is possible to apply an exponent to the incoming data and the operation applied to each channel independently through the relationship in Equation 16.

$$output = input^{power} \qquad (16)$$

The sigmoid module allows for the application of a user adjustable sigmoid-like curve to the incoming data. To create the sigmoid function, the module uses two different power functions to create the sigmoid like shape. The output values are calculated depending on which of the two power functions the input value falls.

The 3X3 matrix multiplication module allows for the coefficients of a 3x3 matrix to be entered from the keyboard or from a text file. Those coefficients, in turn, are used to do a 3x3 matrix multiplication with the input values with form a 1x3 matrix and result in another 1x3 matrix.

4.2.5 Look-up tables

Look-up tables or LUTS are commonly used in many areas of color science and digital image processing. For this research, capabilities for processing 1-dimensional (1D) and 3-dimensional (3D) tables were included.

4.2.5.1 1D Look-up tables

Many operations like negating an image or adjusting its tonescale can be handled through the use of 1-dimensional look-up tables. For this system, 1D LUTS were implemented where the inputs and outputs were digital code values. The input value represents the index in a one dimensional array of values, and the resulting output value is the value stored at that array index as shown in Equation 17.

$$value_{out} = array[value_{in}] \qquad (17)$$

It is possible to use 1D LUTS with floating point data, however, an interpolation scheme is usually required since it might be difficult to fully populate a 1D table with all the possible output floating point values for all the possible input values. This capability was not included in this research, but would be a useful addition to future releases.

A 1D LUT module was created, and it allows for the reading and application of channel independent, 8-bit per channel LUTS where the input values and resulting output values are of type unsigned char, 0-255. Fig. 15 shows the control panel for this module.

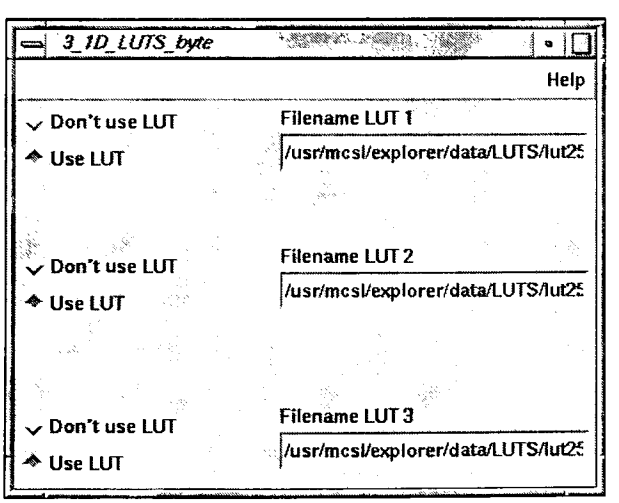


Fig. 15 Module control panel for module *3_1D_LUTS_byte*

4.2.5.2 3D tables

For this research, a simple 3-dimensional look-up table interpolation module has been written along with the two additional modules which provide read and write support for the tables. The 3D LUT interpolator uses linear interpolation to determine the proper output value if a proper value is does not exist in the table. Equation 18 shows the equation used to calculate the resulting value which is based on linear distances from eight surrounding neighbors which form a cube around the point of interest.[5]

$$f(x,y,z) = (1-r)(1-s)(1-t)\, f_{i,j,k} + r(1-s)(1-t)\, f_{i+1,j,k} + (1-r)s(1-t)\, f_{i,j+1,k} + rs(1-t)\, f_{i+1,j+1,k} + (1-r)(1-s)t\, f_{i,j,k+1} + r(1-s)t\, f_{i+1,j,k+1} + (1-r)st\, f_{i,j+1,k+1} + rst\, f_{i+1,j+1,k+1} \quad (18)$$

where r, s, and t are the distances from one of the neighbors to the point and $f_{i,j,k}$ are the neighbors surrounding the point of interest.

The quality of the resultant images processed through the interpolator are based upon the size of the luts used. For this research, different size, factorial designed tables were used. The larger the table, 33^3 nodes for example, produced better results than smaller tables of few nodes.

The interpolator written for this research is very simple. More advanced interpolators could be implemented in the future. Improvements can be made in three main areas. One is the interpolation technique. Other types of interpolation include tetrahedral and bi-cubic.

The next deals with optimizing the table for linear interpolation. This requires the interrogation of an input table to determine how well the table will work with the linear interpolation and if any mathematical transformation of the table might be possible to produce a better table that would reduce the interpolation errors. For example, if a

table shows a good deal of curvature, it may be possible to take the log of the table which might remove the curvature and improve the performance of the linear interpolation. The log table is used to find the value and then the antilog is taken of the result in order to produce the proper output value.

The final technique deals with unequal nodal distributions. The tables used in this research were factorial designs and therefore had equal spacing of the nodes throughout the 3 dimensions of the table. For some algorithms or color spaces, only certain portions of the 3-dimensional space are used. If nodes fall outside of that region, they are wasted. Therefore by understanding how an imaging chain uses the 3-dimensional space, it is possible to unevenly disburse the nodes to make more effective use of the nodes for the space used.

None of these techniques was implemented for this system, but would be an improvement opportunity for a future revision of the system.

4.2.6 Data management, support & analysis

Modules were written for the handling of data of different types, modules designed to provide additional support functions to enhance the system, and modules to help perform different types of analyses on the data.

4.2.6.1 Data management

As had been discussed earlier in this document, the double data type is used to represent the normalized XYZ tristimulus values used throughout the system. Many of the mathematical operations also expect the double data type on input. A user may wish to use these operations on other types of data, however, that data might be represented by

floats or unsigned characters in the case of digital code values. Therefore, a set of modules has been written which translate data from one data type to another. This is done through the casting operation in ANSI C. When going from a smaller data type to a larger data type (*Byte_to_Double*), no data is lost. However, data is lost when values in a larger data type are cast into a smaller data type (*Double_to_Byte*), so care must be taken when performing these operations.

4.2.6.2 Support

Modules have also been provided which provide some additional support functions. Since the entire system uses normalized XYZ values, a module has been created which allows a user to select a white point for most of the common CIE illuminants or enter their own. The *Select_White_Point* module has three parameter outputs, X,Y, and Z, which can be attached to any module requiring a normalizing white point. Fig. 16 shows two instances of the *Select_White_Point* module. The instance on the left outputs the XYZ tristimulus values for the CIE D65 illuminant and the instance of the right allows for the user to enter their own values.

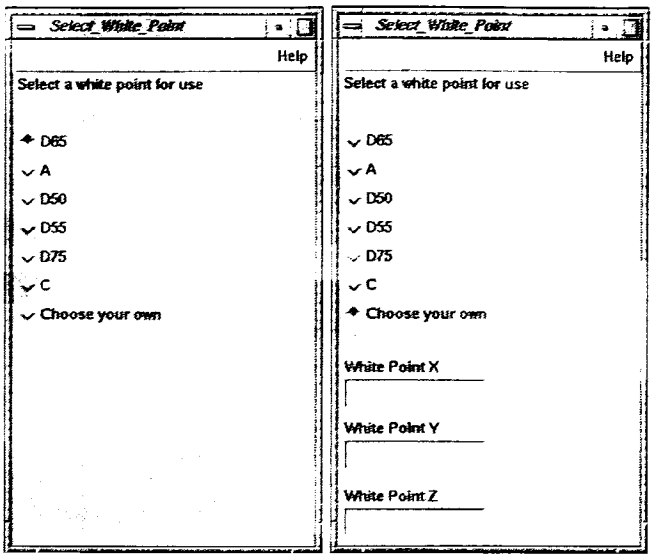


Fig. 16 Module control panels for module *Select_White_Point* with "Choose your own" option enabled and disabled

4.2.6.3 Analysis

Finally, several modules have been written to allow for different types of analyses. The most commonly used module in this set of modules is the *Delta_E* module. This module calculates the average ΔE between two sets of data which are both in the CIELab coordinate space.[3] ΔE_{ab} is calculated through the relationship in Equation 19.

$$\Delta E_{ab} = \sqrt{(\Delta L^*_{ab})^2 + (\Delta a^*_{ab}) + (\Delta b^*_{ab})} \qquad (19)$$

The *Delta_E* module requires two input data sets or images of the same size. It outputs through a widget on the module panel the average ΔE_{ab} for the two data sets. It also has an output port which outputs a single channel Explorer lattice which is the ΔE_{ab} pixelwise for the two images. This image can be displayed or analyzed and provides the user the ability to see what area of an image has the largest ΔE_{ab}.

The *Switch_Image* module has also been provided to help in the analysis of images. Many times, it is useful to be able to compare two images to each other. For example, an original image and one processed through a given imaging chain. Side by side comparisons of images can be accomplished in the Explorer system by using two *DisplayImg* modules. However, it is sometimes difficult to see differences with the both images on the screen. Many times its is useful to be able to view two images separately, but in the same display window. Differences can be easily detected as one image paints over the top of a the one currenlty displayed.

The *Switch_Image* module allows for the input of two different images are Explorer lattices. Through the use of a set of radio buttons, the user can choose which image is outputted to a *DisplayImg* module allowing toggling between two different images.

4.3 Module building

The power of the IRIS Explorer software comes not only in its excellent set of default modules, but the ability of a user to customize and create new modules to add capabilities to the Explorer system that are not available by default. This capability was paramount in the choice of Explorer for this system since the vast majority of the algorithms and image processing capabilites envisioned for this system were not available in the default Explorer module set or in any other commercially available software program.

This section will cover the issues uncovered in learning how to program Explorer modules, a code walk-through for two different modules created for this research, and a

walk-through of using the Module Builder utility for one of the modules created for this research.

4.3.1 Learning to program IRIS Explorer

At the start of this research, no one at the Munsell Color Science Lab had any experience in programming IRIS Explorer. Therefore, a fair amount of time was used learning about the many facets of creating Explorer modules before a single module was built for this system.

Since Explorer has been designed to be able to handle a wide range of applications, there was a large portion of Explorer's capabilities which were not utilized in this research since they were not applicable. This broad capability, however, also acted as a hindrance at the start of this research since time had to be taken to learn enough about the many different facets of Explorer to see which were applicable and which were not. For example, Explorer offers five different intrinsic data types for the passage of data. For this research, only the Lattice type and Parameter type were required for the system planned. However, it was important to understand the uses of the Pyramid, Geometry, and Pick data types in order to design the system since some of the default modules which might have been applicable to the system design expect these data types as input.

NAG and SGI previous supplies a Module Writer's Guide[2] which was a useful book to reference. No other published reference books on module building could be found. The Module Writer's Guide is a useful book, however, the same issue applies as described above. The guide was written to cover all the facets of the Explorer software

which meant the explanation of the parts required for this system was minimized. The explanations given, however, were useful and some sample source code was given to help understand how the modules were to be built. The scope of the sample source, however, was very limited.

The solution to this was to look at the provided source for both the SGI/NAG supported modules and the unsupported modules which were and still are provided. In many cases, however, looking at the different source code files was not necessarily helpful since it was discovered that there are many different ways to author IRIS Explorer modules. The techniques varied based upon the creator and upon the specific application area. For this research, one methodology for module building was adopted and similar techniques were used throughout the entire system. This was done to maximize code re-use and to provide an easy to follow roadmap for future researchers to understand and extend this system. The technique used was not necessarily the best way to create an IRIS Explorer module, however, it was a method which worked well and required a minimum of additional Explorer code in relation to the algorithmic code generated for the user created user function file.

Researching the provided source code, however, did provide a large amount of insight in how to handle specific module building tasks such as setting up widgets and integrating function calls from the Explorer API along with Explorer generated code from Module Builder. The Module Builder utility is very useful and saves a large amount of time in code generation. There are some instances, however, where tradeoffs should be made in allowing Module Builder to generate all of the code versus making some specific API calls in the user function file to produce certain behavior.

A good example of this is the setting up of values in the dial widget which is used in many of the modules for this research. The dial widget provides a dial along with three numbers; a minimum value, a maximum value, and a current value. Fig. 17 shows an example dial widget taken from the *CRT_RGB_to_XYZ_Norm* module.

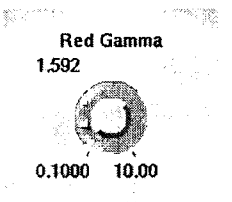


Fig. 17 Example dial widget

One of Module Builder's functions is the Control Panel Editor. This editor allows for the creation of the graphical user interface for a given module. It allows the creator to choose which type of widget to use for a given variable and to set any ranging values on that widget. If the module creator chooses this method, there is no explicit declaration of the values for a given widget type in the user function file. If the creator, however, wishes to be able to override the settings put in place in the Control Panel Editor, it is possible to make an API which sets the values, min and max for example for the dial widget. This explicit call placed in the user function file then overrides any values placed in through the Control Panel Editor. The actual Explorer C API call for setting the minimum and maximum value of a widget is cxWdgtDblMinMaxSet(char *portname, double min, double max).

This method has the benefit of having an explicit call in the user function file which sets up the widget for use and allows for another person to see exactly what the author was intending for the range on the widget. However, setting the values through the Control Panel Editor provides the ability to change the values without having to

recompile the module since those values reside in the module's .mres or module resources file which can be directly saved from Module Builder. To change the settings made through an explicit API call requires a source code edit and a recompilation.

These types of choices are not well articulated in any of the Explorer provided materials and had to be discovered through a combination of reading the provided documentation, source code exploration, and experimentation. For a programmer with a solid background in one of the Explorer supported languages (ANSI C, C++, Fortran), creating Explorer modules should not be very difficult. However, it is in this researcher's opinion that there are many small caveats to Explorer programming which had to be learned by experimentation which required the building of modules in different ways and then testing those modules to determine their behavior.

4.3.2 Code walk through

To help those readers who wish to understand how the user function files for this system were written, this section will provide walk throughs of two different modules written for this research. The first module, *XYZ_Norm_to_CIELab,* shows an example of module which has no widgets and accepts a single lattice on input and outputs a single lattice. The second module, *Channel_Power,* shows an example of a module which uses widgets.

4.3.2.1 XYZ_Norm_to_CIELab

The user function file for this module is named XYZ2CIELab.c. The code is presented in bold italics to offset it from the walk-through comments. All of the source code for this research can be found in Appendix B of this document.

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>
```

Explorer has several different header files which are required to be included based upon the type of module that is being created. In this case, header files for the Lattice data type and the Parameter data type are included along with the header file DataAccess.h which provides calls to access different aspects of the data stored in the Explorer data types. The header files and the functions found inside of them can be found in the Explorer API manual.[4]

```
void XYZ_Norm_to_Lab(cxLattice *in, cxLattice **out)
```

Standard C function declaration. Since the lattice *out* has not been created yet, a double pointer is required.

```
{

    double *a;
    double *b;
    int i,j;
    cxErrorCode err;
```

Variable declaration includes pointers which will be used to access the data in the input and output lattices, counter variables *i,j* for use in looping through the lattices, and another intrinsic Explorer data type used in reporting errors in module execution to the screen.

```
    /* create the new lattice */
    *out = cxLatDataNew(
            2,                  /* number of dimensions */
```

```
    in->dims,                /* dimensions array */
    in->data->nDataVar,      /* number of data variables */
    cx_prim_double);         /* data type */

/* use the same coordinate space as the input lattice */
cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);
```

Explorer API call necessary to create the new lattice *out*. Through this call, Explorer reserves the necessary area in its shared memory arena for the data portion of this new lattice. A similar call is not needed for the input lattice since it had to have been created in a module previous to this one in order for it to be passed in. The Lattice data type is composed of two different arrays. One array holds the data for the lattice. The other holds the values associated with the coordinate system for that particular lattice. In the case of this research, all of the Lattices are assumed to be uniform. Therefore, the second API call below simply copies the coordinate system of the incoming uniform lattice into the new output lattice.

At this stage, it is also important to understand the different variables used in the *cxLatDataNew* API call. First is the number of dimensions for the data. In the case of this research, images are the main usage. Images are two dimensional, so the first value is set to 2. The dimensions array, in turn, is an array that contains the actual size of the data array based upon the number of dimensions. In this case, the dimensions array has two elements of ANSI C type *int* (integer) and the output dimensions are being set to match the input dimensions array. The next variable which has to be set is the number of data variables for each location in the two dimensional array. For this research, this value is normally set to three since each pixel location usually has three values such as RGB, XYZ, LAB, YCC, and etc. In this code section, the value will be copied from the input lattice. In essence, the data array for this particular lattice is really three

dimensional, however, Explorer uses a slightly different way of creating that data array. The final variable to be set is a ENUM type which determines the C data type for the data in that array. In this piece of code, the values are set to be doubles or the equivalent Explorer ENUM value of *cx_prim_double.*[2]

```
/* extract the data pointers */
cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
```

In the variable declarations, two double pointers were declared. The above two API calls are used to set those pointers to the memory location where the data arrays for both the lattice *in* and lattice *out* are located in memory. Since the pointers were declared as doubles to begin with, they are cast here as type void. It is possible to change this around if the module writer wishes to have flexibility in the type of data used. It is possible to declare void pointers in the variable declaration and then cast them here to match whatever data type is needed. This is very useful for functions which need to be able to accept lattices in different C data types such as doubles, floats, ints, and unsigned characters.

```
for ( i = 0; i < in->dims[1]; i++ )
{
        for ( j = 0; j < in->dims[0]; j++ )
        {

            if(a[1] > 0.008856)
            {
                b[0] = (116.0 * pow(a[1], (1.0 / 3.0)) - 16.0);
            }
            else
            {
              b[0] = (903.3 * a[1]);

            }
```

```
                    if(a[0] > 0.008856 && a[1] > 0.008856)
                    {
                        b[1] = (500.0 * (pow(a[0], (1.0 / 3.0)) - pow(a[1], (1.0 / 3.0))));
                    }
                    else if(a[0] <= 0.008856 && a[1] > 0.008856)
                    {
                     b[1] = (500.0 * (((7.787 * a[0]) + (16.0 /116.0)) -
                            (pow(a[1], (1.0 / 3.0)))));
                    }
                    else if(a[0] > 0.008856 && a[1] <= 0.008856)
                    {
                      b[1] = (500.0 * ((pow(a[0], (1.0 / 3.0))) -
                            ((7.787 * a[1]) + (16.0 /116.0))));
                    }
                    else
                    {
                        b[1] = (500.0 * (((7.787 * a[0]) + (16.0 /116.0)) -
                            ((7.787 * a[1]) + (16.0 /116.0))));
                    }



                    if(a[1] > 0.008856 && a[2] > 0.008856)
                    {
                        b[2] = (200.0 * (pow(a[1],  (1.0 / 3.0)) - pow(a[2],  (1.0 / 3.0))));
                    }
                    else if(a[1] <= 0.008856 && a[2] > 0.008856)
                    {
                     b[2] = (200.0 * (((7.787 * a[1]) + (16.0 /116.0)) -
                            (pow(a[2], (1.0 / 3.0)))));
                    }
                    else if(a[1] > 0.008856 && a[2] <= 0.008856)
                    {
                      b[2] = (200.0 * ((pow(a[1], (1.0 / 3.0))) -
                            ((7.787 * a[2]) + (16.0 /116.0))));
                    }
                    else
                    {
                     b[2] = (200.0 * (((7.787 * a[1]) + (16.0 / 116.0)) -
                            ((7.787 * a[2]) + (16.0 / 116.0))));

                    }

                        a += 3;
                        b += 3;

                }
            }

}
```

This large nested for loop goes through the data array and calculates the CIELab values from the incoming normalized XYZ values. The data variables at each location in the two dimensional image are accessed as a one dimensional array, *a[x]* where x is equal to 0 for the normalized X tristimulus value, 1 for the normalized Y tristimulus value, and 2 for the normalized Z tristimulus value. The resulting CIELAB values are accessed through a similar one dimensional array, *b[x]* where x is equal to 0 for L*, 1 for a* and 2 for b*.

After the calculations are made, the pointers *a* and *b* are incremented by three since there are three data variables.

4.3.2.2 Channel_Power

The *Channel_Power* module applies an channel independent exponent to the input values to create the resulting output values. The source code file is named power.c.

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>
```

Same header files declared as in the last function.

```
void Power(cxLattice *in,  cxLattice **out,
                    double red_power, double green_power, double blue_power)
```

Standard ANSI C declaration which is slightly different from the previous program. For this module, there are three widgets, dial widgets, which allow for the entry of an exponent for each channel. Widgets only apply to single data values like the parameter data type. In this case, the values for each exponent are of C type double and are not

declared as Explorer type cxParameter. This is possible and can be a point of confusion for the novice module builder since it is also perfectly legal to have these values declared as cxParameters. In either methodology, however, it is required that if a widget is to be used, an input port must be declared for that variable. For that input port to be declared, it is necessary to include that variable in the parameter list of the user ANSI C function file.

```
{

 cxErrorCode err;
 double *a;
 double *b;
 int i, j;



        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,               /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);        /* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
```

Same code as found in previous module showing the code reuse possible if a single module building method is chosen.

```
if(red_power == NULL)
{
      red_power = 1.0;
}
if(green_power == NULL)
{
      green_power = 1.0;
}
if(blue_power == NULL)
```

```
{
    blue_power = 1.0;
}
```

When a module has single values or parameters that it wants to associate with widgets, there is a chance that the values may come in from an input port or might be directly inputted through the widget(s) on the modules user interface. As soon as a module is initiated in the Map Editor, the Explorer firing algorithm[2] executes the module which can create problems if the code is not properly set up to handle this situation since there may be no value associated with a given variable.

This piece of this user function file checks to see if the variables are equal to NULL which indicates that the variable has not been assigned a variable either through the user interface or through an input port. If it sees NULL, it then sets the variables for the red, green, and blue exponent to 1.0 for this module. The value assigned depends on the algorithm being implemented. In this case, a value of 1.0 was used since any number raised to the power of 1.0 is equal to the inputted number. Therefore, the module will not affect a change in the data until the user changes the value.

It is also possible in these types of cases to use the check for NULL and then use a return statement instead of an assignment statement. This allows no value to be set until a user directly inputs a value and prevents the rest of the user function file from being executed until a value is set. An example where this might be useful is where the value being set is being used as a multiplier in a denominator of a fraction. If the module was allowed to execute with a value of NULL which is equivalent to zero, the module would core dump on a divide by zero error.

```
cxInWdgtDblSet("Channel 1 Exponent Value", red_power);
cxInWdgtDblSet("Channel 2 Exponent Value", green_power);
cxInWdgtDblSet("Channel 3 Exponent Value", blue_power);
```

Once the value has been determined for each of the three variables, *red_power, green_power,* and *blue_power,* three Explorer API calls are made which assign those variables to the the proper input port. In turn, the input ports for the parameter datatype are then linked to the user's choice of widgets. This relationship will be explained in more depth in the following section on how to use Module Builder.

The code here also shows that explicit declaration of the variables are being made in the user function file instead of through the Control Panel Editor in Module Builder.

```
/* loop over the data elements */
for ( i = 0; i < in->dims[1]; i++ )
{
        for ( j = 0; j < in->dims[0]; j++ )
        {

        b[0] = pow(a[0], red_power);

        b[1] = pow(a[1], green_power);

        b[2] = pow(a[2], blue_power);


        a += 3;
        b += 3;

        }
}

}
```

Identical in concept to the previous code walk-through, this snippet loops through the incoming lattice, calculates the exponentiated value, and places the new value in the output lattice. Finally, it increments each lattice pointer by three for the number of data variables and ends the function.

4.3.3 Using Module Builder

The Module Builder utility supplied by NAG with IRIS Explorer is a very compact application program which offers a lot of power for those who wish to create new IRIS Explorer modules or modify existing modules. Module Builder allows a user to concentrate on being an algorithm developer and not have to worry as much about many of the programming aspects usually associated with working with images like memory management. However, a solid foundation in programming is very useful in understanding what Module Builder is doing for the creator and when the creator should take control of the code generation.

The Module Builder utility comes in to play when the creator has completed a user function file and wishes to create an IRIS Explorer module. In the following section, Section 4.3.3.1, a step by step walk-though is done for the Channel_Power module based on the power.c user function file which was walked through in Section 4.3.2.2. This should help first time users understand the different parts of using Module Builder to set up a module and then compile it. However, before invoking Module Builder to create a module, some things need to be done to insure that Module Builder will do what the module creator expects.

4.3.3.1 Considerations before invoking Module Builder

The most important concept to consider before invoking Module Builder is the setting of the EXPLORERUSERHOME environment variable. This environment variable, set through the UNIX setenv command, tells Module Builder where to build

your module upon compilation. This does not necessarily have to be the same as the location of your user function file, but it can be. If this variable is not set, its default location is the directory /usr/explorer for SGI machines. The problem with the default location which can also be a benefit is usually this directory has its file write permissions set only for the root owner of the system. This means that all users other than root can not write to that directory unless the root owner has changes the write permissions. Therefore, Module Builder will not be able to compile the program since it can not write the different C code files it creates along with the compiled object and executable files for the module.

For the development of the code for this research, each module built was assigned its own directory. For a few modules, a common directory was used. This was done to help keep all of the code separate during development. This is not a required practice or possibly not the best practice for some projects, but it was useful in the case of this research.

When a user function file was completed, the EXPLORERUSERHOME environment variable was set to the directory where the user function file was located. For example, the code for the *Channel_Power* module which has been discussed in Section 4.3.2.2 was located in the directory /usr/people/crh3821/Explorer/Thesis/Power. Therefore the call required to set the variable for Module Builder was as follows:

setenv EXPLORERUSERHOME /usr/people/crh3821/Explorer/Thesis/Power

This was entered through a standard C shell window on the SGI systems. This same command also works for most of the other UNIX versions of EXPLORER and has been verified to work by this author on the Sun Solaris 2.5 implementation. A similar type

command structure is used for the Windows NT version of Explorer, however, the location where the variable is set is located under the Windows Control Panels.

On the Silicon Graphics machines, it is possible to have icons for Explorer, Module Builder and other applications on the user's desktop. There is an issue in using the icons to launch Module Builder by double clicking on the icon versus launching Module Builder by typing *mbuilder* in a UNIX shell window. When using the icons, Module Builder uses the environment variables which are set through the user's *.cshrc* file in the case of a C shell. It does not pay attention to any environment variables set through a C shell window. Therefore, if the user in a C shell window sets EXPLORERUSERHOME and then invokes Module Builder through the icon, Module Builder will use the default EXPLORERUSERHOME definition. However, if the user sets EXPLORERUSERHOME through a C shell window and then invokes Module Builder by typing *mbuilder* in that same window, Module Builder will use the new set location for EXPLORERUSERHOME. Therefore, it is possible on a single desktop to have different instantiations of Module Builder from multiple C shell windows all pointing at different setting of EXPLORERUSERHOME.

Should a module creator decide to have a single location for building modules, the EXPLORERUSERHOME variable can be set through their *.cshrc* file, and every instantiation of Module Builder by either means from that point on will reference the location defined in the user's *.cshrc* file. This the left to the system creator to decide what method works best. One compromise position which works is to set EXPLORERUSERHOME to a known location through the *.cshrc* file and then change it through a C shell if a different location is needed for a specific module build.

There are several other environment variables which can be used during module building and installation are covered in the IRIS Explorer Module Writer's Guide. One example of these other variables is CXBUILDSHARED which tells Module Builder to use shared libraries instead of static libraries for the Module Control Wrapper and API library.

For basic module building, however, the module creator need only be concerned with setting the EXPLORERUSERHOME environment variable.

4.3.3.2 Invoking and Preparing to Use Module Builder

As covered in the previous section, it is up to the user to decide how to invoke Module Builder; either through the icon or through a C shell window. Once it is started up, the Module Builder window will appear on the screen, and the module creator can start working on a new module or editing a previous module.

Along with the different compiled executables, all IRIS Explorer modules have an additional file called the module resources file where different parameters about the module are stored. These files have the *.mres* extension and share the name of the module. For example, the *Channel_Power* module has a module resources file named *Channel_Power.mres*. Module Builder uses the *.mres* file to store and set-up all of the parameters for building a module. Explorer itself uses this file as the master file to tell it what to open for a given module and how to set-up its user interface. This file is analogous to the app-defaults file used in Motif programming[15] where different attributes of a program can be set-up through this file allowing a user to change some parts of an

application without the need for recompilation. These changes usually revolve around the user interface, but can go deeper.

To work with an existing module in Module Builder, a user should go through the File menu to the Open command and then proceed to open the proper *.mres* file for the module they wish to work on. This will setup Module Builder properly. The user should not try to open any of the other files since Module Builder will not know how to use them.

This same procedure is also used in Explorer for opening a module which may not be in the Module Librarian. It is possible through the same File menu and Open command to launch a module in the Explorer Map Editor by simply selecting the proper .mres file. This is very useful during module development since the Module Librarian may not be set up to point to the location of the newly created module, and a user may not want to have to quit Explorer, modify the .explorerrc[1] file if needed, and then restart Explorer to let the Module Librarian see the module.

When creating a new module, it is not necessary to open anything up. The user can simply start entering parameters into Module Builder. After entering some of the parameters, it is possible to use the Save Module Resources option under the File menu to save the new *.mres* file for that module. Like working in any type of application, performing a save every so often insures that time will not be lost should a system malfunction take place. It is also useful if a module creator wants to stop midstream on creating a module, save the current work, and return to it at a later date.

In either case, a user is now ready start the process of setting up all of the necessary parameters in Module Builder to build an Explorer Module. Since Module

Builder is a relatively compact application, but requires some understanding, the following section will perform a walk-through of the Module Builder setup for creating the *Channel_Power* module.

4.3.3.3 Module Builder walk-though - *Channel_Power*

For this section, images of the different Module Builder command windows will be shown along with explanations of the contents. To begin, Fig. 18 shows the main Module Builder window.

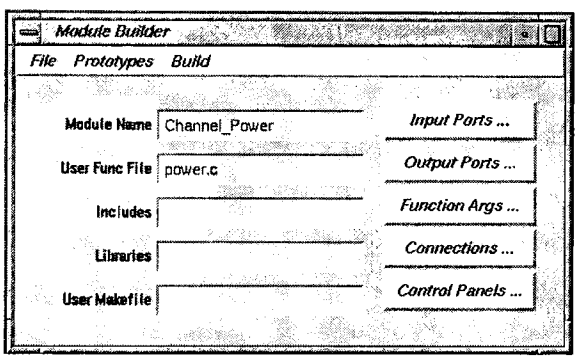


Fig. 18 Module Builder utility main control panel window

There are three different pull-down menus, File, Prototypes, and Build. There are five button widgets which give the user access to the different submenus which will be covered later in this section. There are five different text type-in widgets for entering different parameters including the module name, user function filename or names if other C files are also used, an special header files, any special libraries, and the name of a user defined makefile.

Module Builder will create its own makefile for the given module, however, it is possible for a user to create their own or use a modified version of the automatically generated one. Since Explorer modules may be built upon custom libraries of functions,

Module Builder provides a location to enter the names of those libraries. Associated with those libraries or possibly with additional code beyond the main user function file, may be some specific header files. The names of these should be entered in the space provided. Finally, it should be noted that not all of the code for a user function file needs to go into a single file. ANSI C programming allows different functions to be placed into different files and then combined during compilation. The same is true when using Module Builder. There does need to be one "main" user function file which gets defined and is listed first. However, other helper files such as different image processing subroutines should also be listed in the User Func file window since this is how Module Builder will know what needs to be compiled. Although it will not be covered here, a good example of having multiple C files along with the user function file is the *XYZ_Norm_to_Hunt93* module. Opening the *XYZ_Norm_to_Hunt93.mres* file for this system in Module Builder will show the setup for using multiple files.

In this example, only the Module Name and User Func file type in slots are used since there are no special header files, libraries, or makefiles used. Also, all of the code necessary for this module resides in a single user function file which was described earlier, so there is only a single name. It should also be noted here that the filename entered, *power.c*, under the User Func file menu does not have its full pathname which would have been during the development */usr/people/crh3821/Explorer/Thesis/Power/power.c* . Since Module Builder was invoked from the directory where the source code was located, the full path was not needed. However, if a user is not going to go through the trouble of going to the proper

directory before invoking Module Builder, it is advisable to enter the entire pathname of the user function file to be worked on.

With all of the proper information entered, the next step in setting up Module Builder is to click on and go to the Input Ports window where the different input ports if any will be defined. Fig. 19 shows the Input Ports window for this module.

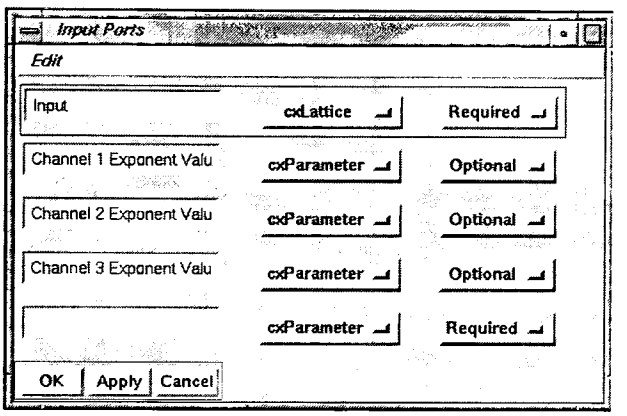


Fig. 19 Module Builder input ports control window

In the case of this module, four different input ports are being defined. The Input Ports windows is very simple and has only three required parameters to be set for each port defined. These include the name of the port entered in the text type in slot, the type of data for that port which is selected from a menu which appears when the middle button is depressed, and finally whether the port is required or optional.

Explorer uses a specially designed firing algorithm which is covered in the Module Writer's Guide to tell Explorer when to execute a module. If a module has input ports and they are set as *Required,* Explorer will not fire that module until it senses that all of those input ports are connected to another module and are being fed data. Similar to the situation described in the code walk-through, this prevents a module from

executing before pertinent data values are set and prevents core dumps or unnecessary time waiting for a module to complete executing on invalid data.

This system, however, does not work for modules where input data may come directly from a widget on that module and not from an input port. In this case, it is necessary to set those input ports where values may be entered from widgets to *Optional.* This is the case for the *Channel_Power* module where the three input ports for the three different exponent values are set to *Optional.* This is another reason why, as was described in the code walk through, it is sometimes necessary to add code to check to see if a given input port has been set to a value or if its value is NULL. When an input port is set to *Optional*, the module will fire and execute even if those ports are not connected to another module and receiving data.

The first input port defined is named *Input.* This is the only one which is required since it is the incoming lattice data. Without the lattice data coming in, the module has nothing to process. The type is set to cxLattice which was chosen from the menu which appears when that button is clicked. Unlike the cxParameter type which has no additional input requirements, the cxLattice type has an additional control window which is invoked when cxLattics is chosen.

Fig. 20 shows the Input Port/Lattice Constraints menu.

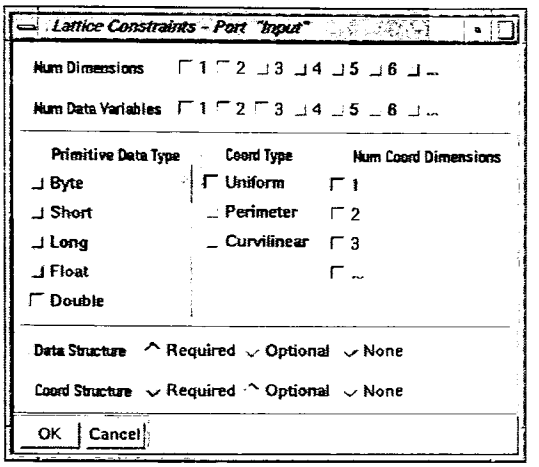


Fig. 20 Module Builder Lattice constraints window for input port "Input"

The Lattice Constraints menu allows the user to tell Module Builder what type of Lattice it should be expecting. In the case of the *Channel_Power* module, the Lattice has two dimensions, so both *1* and *2* are chosen as valid for **Num Dimensions**. The Lattice also has three data variables, so *1*, *2*, and *3* are chosen as valid for **Num Data Variables**. Since the incoming data is in normalized CIE XYZ values which are of type double, *double* is checked as valid for **Primitive Data Type.** Since images are uniform lattice structures, *uniform* is checked for **Coord type.** Finally, all of the values for **Num Coord Dimensions** are selected as valid. Since the coordinate system is not being used in this system and since Explorer will not allow modules with dislike constraints to link to one another, no restrictions were placed on this variable. If the coordinate system was being used, a more definitive choice may be necessary. This is also shown in the final two selections. For **Data Structure,** *required* is selected since the system needs the image data stored in the data structure. However, **Coord Structure** is set to *optional* since the system does not require a coordinate system to be there, but does not want to reject a lattice if a coordinate system exists.

Once the constraints are properly entered, clicking *OK* will apply the changes and return to the Input Ports menu. Clicking *Cancel* will discard any changes and return to the Input Ports menu. In either case, once the module creator has made all of the selections, it is necessary to exit the Input Ports menu. To accept the changes and exit, click *OK*. To apply the changes, but not exit, click *Apply*. Finally, to cancel any changes and exit to the main menu, click *Cancel*.

The next step in working with Module Builder is to define the output ports through the *Output Ports* button on the main Module Builder control window. Fig. 21 shows this window.

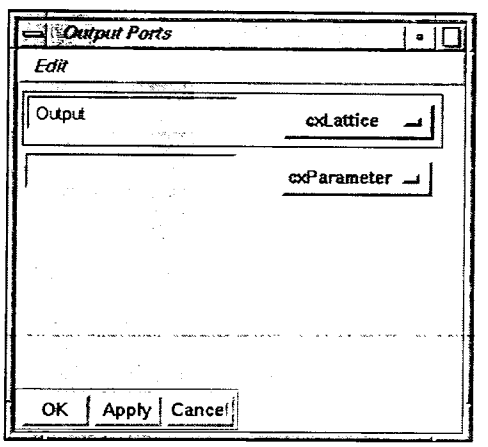


Fig. 21 Module Builder Output Ports control window

Slightly different from the Input Ports window, the Output Ports window only has two sets of widgets. Since it is not required to define an output port as required or optional, that set of widgets has been removed. It is only necessary to define the output port names and the data type. In this case, the output port has been named *Output* and the type has been set to *cxLattice*. It should be noted here that Module Builder and Explorer has a feature which automatically provides all of the input ports which are connected to widgets as output ports. It is not necessary for the module creator to do

anything. They will simply appear when the output port control pad is clicked on when the module is executed in the Map Editor window. All of the defined output ports will be listed followed by a line and then outputs for all of the input ports will be listed. This feature is very useful for testing and future expansion.

Just like the Input Ports window when selecting the *cxLattice* type, a Lattice Constraints window appears. Fig. 22 shows this window for the output.

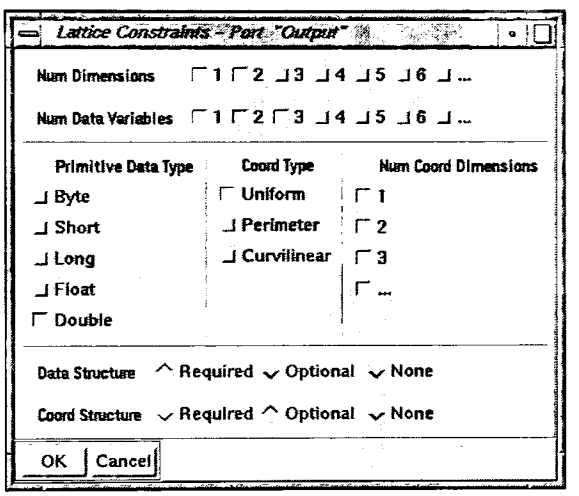


Fig. 22 Module Builder Lattice constraints window for output port "Output"

Since the output lattice is the same as the input lattice, the same constraints are set. This is not always true. For example, a module like *sRGB_to_XYZ_Norm* accepts 8-bit per channel byte (unsigned char) data on input and outputs double data. In that case, the Input Port Lattice Constraint would be set to *Byte,* while the Output Port Lattice Constraint would be set like this case to *Double.*

Once completed and returned to the main Module Builder window, the next step is to click the *Function Args* button to bring up the Function Arguments window. This is the window where the module creator tells Module Builder about the user function file it has created. This window will change depending on the programming language being

used. The window shown in Fig. 23 is the window for those module creators using ANSI C which this system used.

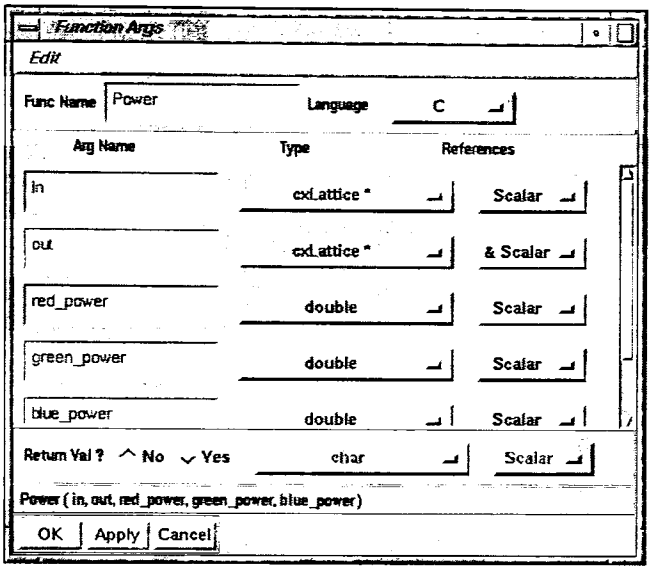


Fig. 23 Module Builder Function Args control window

The first step in defining the function in the Function Args window is to type in the name of the C function used in the user function file. In this case, the name is *Power*. The next step is to start to enter the variables used in the parameter list of the user function file. For purposes of comparison, the C function prototype taken from power.c is as follows:

```
void Power(cxLattice *in,  cxLattice **out,
                double red_power, double green_power, double blue_power)
```

For the first variable, *in*, which is of type cxLattice, the name is entered, then the type, and finally it is necessary to define whether the variable is a scalar or an array and whether it is an input variable or output variable. For *in*, the type is set to **cxLattice*, and it is defined as a scalar since it is a single entity. A variable which, for example, is a character array that holds a filename would be defined as an array. Since *in* is an input

lattice, *Scalar* and not *&Scalar* is chosen. For the next variable, *out*, which is the output lattice, it should be noted that it is defined as *&Scalar* since it is an output variable.

Since the function *Power* returns void, *No* is selected for **Return value?** since there is none. Below **Return value?** is the function prototype which should match up with the C function in the user function file. The only different is the lack of the data types before the variables. This function prototype will not appear until the *Apply* button is clicked, and it will not be updated unless *Apply* is clicked again. Clicking *OK* before clicking *Apply* will still tell Module Builder to accept the new changes. The display of the function prototype is there for double checking to make sure the module creator has the variables in the proper order. Clicking *Cancel* will again discard any changes and return the user to the main Module Builder window.

With the function arguments sets, it is then time to set up the connections between the input ports, the user function file, and the output ports. This is done through the Connections window which is brought up when the *Connections* button is clicked. Fig. 24 shows the connections window for the *Channel_Power* module.

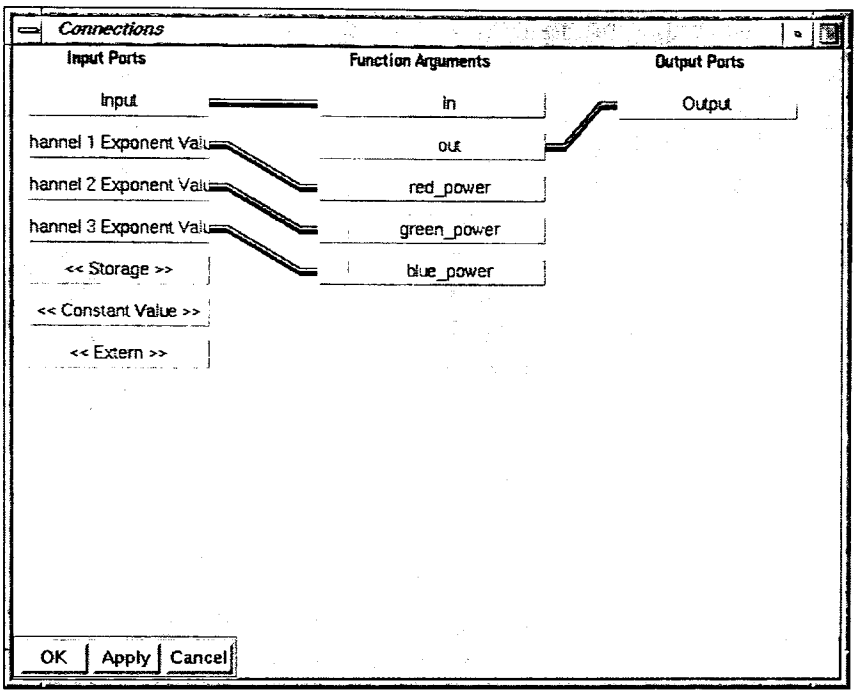


Fig. 24 Module Builder Connections window

The Connections window is very similar in concept to the Explorer Map Editor and connections are made in a similar way to connections made between modules. By placing the cursor over one of the boxes and holding down the right mouse button, a pop-up connections menu appears. The menu differs whether you are on an input/output port or on a variable listed for the user function, and the menus for the input and output ports differ depending on what intrinsic Explorer data type was chosen for that particular port in the Input Ports and Output Ports windows.

The connections menu for each of the intrinsic data types allows the user to connect the entire data structure for that type or different parts of the data type. In this module for example, the entire Lattice data structure is linked from the input port *Input* to the function argument *in*. The connection is shown, but the details apparent when the

connection menu is present are not. It was not possible to achieve screen captures of the actuated menus for this document, however, the connections menus are shown in the Explorer Module Writer's Guide.

It would also be possible, if desired, to link just the data structure portion of the Lattice data type up. This is what is done for the three input ports for the exponent values. The Explorer Parameter type is a data structure with two pieces. It has a piece which encodes the type of data, and a piece which holds the actual data value. For this module, we have defined the input ports to be of type cxParameter, however, the variables in the user function file have been defined as ANSI C doubles and not parameters. This works, but requires some attention when making the connection. To make the proper connection, it is necessary to only connect the data portion of the input port Parameter type to each respective function argument. Since the function arguments are doubles, it is incorrect to connect the entire cxParameter structure since the function argument can not store all of the data in that cxParameter data type. However, it can store just the value stored in the cxParameter data type, and that is what is connected for each of the exponents. This is a very important piece to learn in module building and can save a lot of time.

Although not used, but should be noted are the *Storage, Constant Value, and Extern* input ports. These are always there and are used for different purposes which are described in the Module Writer's Guide[2], but will not be described here. Their functionality was not required to create this system.

Once all of the connections are complete and the proper button clicked to return to the main Module Builder window, there is one final step to complete the setup of Module

Builder. This final step is to configure the user interface through the Control Panel Editor. This step is only necessary for modules which have widgets to be setup. Modules such as *XYZ_Norm_to_CIELab* do not require any work in the Control Panel Editor, and Module Builder will flag the user as such if the button is clicked.

The Control Panel Editor is composed of two windows. One is the editor itself and one is actual module control panel which is under modification. Fig. 25 shows the two windows.

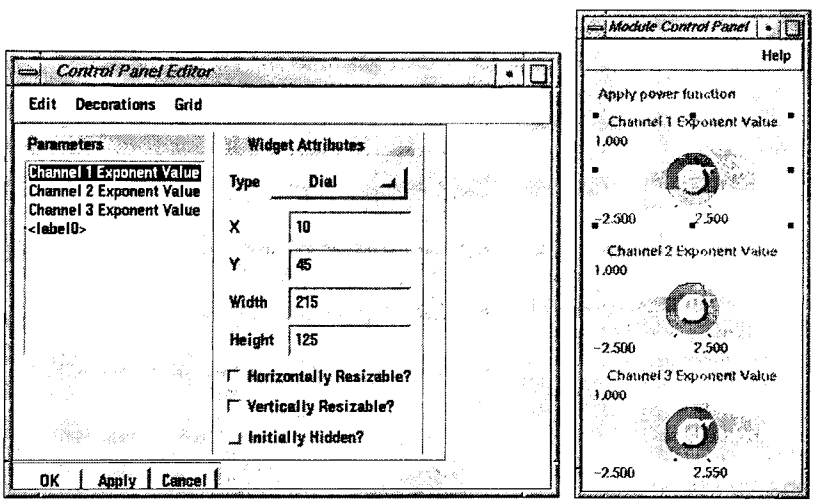


Fig. 25 Module Builder Control Panel Editor window and Module Control Panel window

The Control Panel Editor shows the list of possible parameters which can have widgets associated to them. Depending on how the module is written, it may not be necessary to associate a widget with every parameter. Under the **Type** menu under **Widget Attributes**, there is a *None* selection for just that purpose.

For the *Channel_Power* module control panel shown here, there are three parameters available for associating widgets, and all three have had dials associated with them. In Fig. 25, *Channel 1 Exponent Value* is selected and it is possible to see the

associated widget for that parameter highlighted with small squares in the Module Control Panel window. In the Module Control Panel window, it is possible to change the size of the widget as well as set the different values as previously discussed. In this case, the values have been set here. Also, the code itself has the NULL checking mechanism and the API calls which will change the current set value at run time. As a default, the current value is set to 1.0 in the Editor as well.

There is also an entry in the **Parameters** list named *Label0.* It is possible to add labels and decorations like horizontal and vertical lines to the module's control panel to help improve the user interface. The decorations are available through the **Decorations** menu at the top of the Control Panel Editor.

Once the module control panel is completed and the appropriate button clicked to accept or cancel the changes, all of the values are now defined for Module Builder. The next step is to build the module. Before building, it is recommended that the user do a save of the module resources through the **File** menu. This insures all of the work to setup Module Builder will be saved in case anything should go wrong during building.

4.3.3.4 Building a module in Module Builder

With all of the values entered into Module Builder, it is now time to build the module. This is done through the **Build** menu on the Module Builder main window. Under **Build**, there are several options. The first, **Options...,** defines some simple build options for Explorer. Typically the default setup for this is fine, and no adjustments are needed. However, a user may want to check to see if the following default options are set. Table IV shows the options.

Table IV Default build options for Module Builder using ANSI C

Option	Setting
Write Wrapper Code	Yes
MCW type	Default
Link module with	C
Alternate executable	No
Additional files to install	Leave blank
Loop controller status	None
Output window	Map Editor Window

The next value on the Build menu, **Hook Funcs...**, can be ignored for this system since they are not used. However, the next two entries, **Build** and **Build and install**, are very important. The last entry, **Update document**, is also important once a module is built and running since it is used to tell Module Builder that the modules documentation has been updated by the module creator. Module Builder will then use that updated *.doc* file as the file to create the *.help* file for that module and the man page for that module when **Build and install** is next selected for that module. The **Build** option will simply build the module in the location set by the EXPLORERUSERHOME variable. This option is useful for building and testing modules.

On the other hand, the **Build and install** option adds additional functionality and should be used once the module is running. **Build and install** creates additional documentation and man page files as well as it creates a **modules** directory in the location where EXPLORERUSERHOME points and installs the module executable, module

resource file, module documentation file, and the module credit file in that modules directory while also building the module in the location of the source code.

For this system, **Build** was used initially to compile, build, and the test the individual modules in their own directories in this researcher's home account. Once the modules were tested and were satisfactory, a set of directories was set up to allow others to access the modules within the Munsell Lab and the **Build and install** option was used. This setup is very similar to the setup used by Explorer itself in its */usr/explorer* directory. The special */usr/mcsl/explorer* directory had a *modules* directory and a *src* directory where all the source code was stored in different subdirectories. EXPLORERUSERHOME was set to */usr/mcsl/explorer* and the build/installs were done on the modules in the different directories under */usr/mcsl/explorer/src*.

The results of this process placed all of the module executables and resource files in a single directory. This simplifies access for users since it only requires one special entry in their .explorerrc configuration file pointing to */usr/mcsl/explorer/modules* instead of many different entries to lots of different subdirectories. Different module developers may wish to setup their systems different from this one and build their modules in different ways. The method outlined here seemed to work well for this system, but may not be optimal for other environments.

When either **Build** or **Build and install** is chosen, Module Builder invokes the compilers and compiles the module. Module Builder will put up an error message box if the compilation is not successful. The output of the compiler will be displayed in the C shell window where Module Builder was executed from or in the Console window if EXPLORERUSERHOME was set through the *.cshrc* file and Module Builder executed

from an icon. This output is very important since it will show the module creator where errors in the code are if there are any.

Once compiled successfully, a module can be opened and tested in Explorer. As with any type of programming, compilation is not a sign that the module will run in Explorer. Only after instantiating the module in the Map Editor, making the proper connections, entering the proper settings, and viewing the output can the module truly be validated. Just instantiating it in the Map Editor is not itself a test since the module may need ports wired to it before it will fire and execute. Until it sees those ports connected, the module and the user function have not been executed, so any problems may not be apparent without hooking up all the proper connections, setting any widgets to valid values, and checking the output.

If the module tests successfully, the module documentation can be edited by editing the *.doc* file in any text editor and going through the process outlined earlier in this section of selecting **Update document** from the **Build** menu and then doing a new build and install by selecting **Build and install** from the **Build** menu. This then completes the module building process.

5. Conclusions

A system of modules for NAG IRIS Explorer has been built for this research to perform different device independent color imaging algorithms. The system built inside the NAG IRIS Explorer software has proven to be flexible, easy to use, expandable, and powerful enough to deliver the performance necessary for interactivity while providing capabilities for processing larger images for use in different research projects. The

Explorer software provided a good core set of functionality for performing many of the basic image and data handling tasks while providing an expansion mechanism to build in the device independent algorithms.

Some limitations of the Explorer software have also been identified and solutions sought. The biggest drawback to Explorer which is also a drawback to other packages which use a module based processing paradigm is the large amount of memory required when processing large images through large imaging chains. Several solutions were found. Some of the solutions including the concept of making a module "forget" are useful, however, they lessen some of the benefits of the module based processing paradigm. One solution, 3D luts, however, proved to be an excellent option which allowed small data sets to be processed through the imaging chains while allowing a single module to process large images through the resulting 3D luts.

Finally, the process of building new IRIS Explorer modules has been documented to help future creators of IRIS Explorer modules. The provided documentation for Explorer is very good, however, Explorer has a very large function set with much of Explorer's capabilities falling out of the scope of this research. Therefore, the applicable documentation is therefore limited since the documentation covers all facets of Explorer equally. Additional time had to be spent in this research to learn Explorer and Explorer module building. The documentation provided in this research should allow for a shorter learning curve for new users of the system.

Overall, IRIS Explorer is a very capable package with a great deal of power. Some of its capability was tapped for this research and more of its capability is applicable to other portions of color science research. The package is not perfect, but it provided

the proper framework to allow the system outlined in this document to be built and perform to expectations.

6. References

[1] *NAG IRIS Explorer User's Manual Version 3.5*, NAG, 1996.

[2] *NAG IRIS Explorer Module Writer's Guide,* NAG, 1996.

[3] *Colorimetry,* 2nd ed., CIE Publ. No 15.2, Central Bureau of the CIE, Vienna, 1986.

[4] *NAG Explorer API Reference Manual,* NAG, 1996.

[5] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C,* Cambridge, 1992.

[6] *International Color Consortium(ICC) Profile Format,* Version 3.2, ICC, 1996.

[7] R.S. Berns, R.J. Motta, and M.E. Goraynski, "CRT Colorimetry, Part I: Theory and Practice" *Color Res. Appl.* **18**, 299-314 (1993).

[8] R.S. Berns and M.J. Schyu, "Colorimetric Characterization of a Desktop Drum Scanner Via Image Modeling," *IS&T/SID 2nd Color Imaging Conference,* Scottsdale, 41-44 (1994).

[9] M.D. Fairchild, R.S. Berns, A. Lester, and H.K. Shin, "Accurate Color Reproduction of CRT-Displayed Images as Projected 35mm Slides," *IS&T/SID 2nd Color Imaging Conference,* Scottsdale, 69-73 (1994).

[10] R.W.G. Hunt, "An improved predictor of colourfulness in a model of colour vision", *Color Res. Appl.* **19**, 23-26 (1994).

[11] M.D. Fairchild and R.S. Berns, "Image color appearance specification through extension of CIELAB, *Color Res. Appl.* **18**, 178-190 (1993).

[12] M. Stokes, R. Motta, M. Anderson, S. Chandresekar, "A default color space for the Internet: sRGB", Version 1.10, White paper (1996).

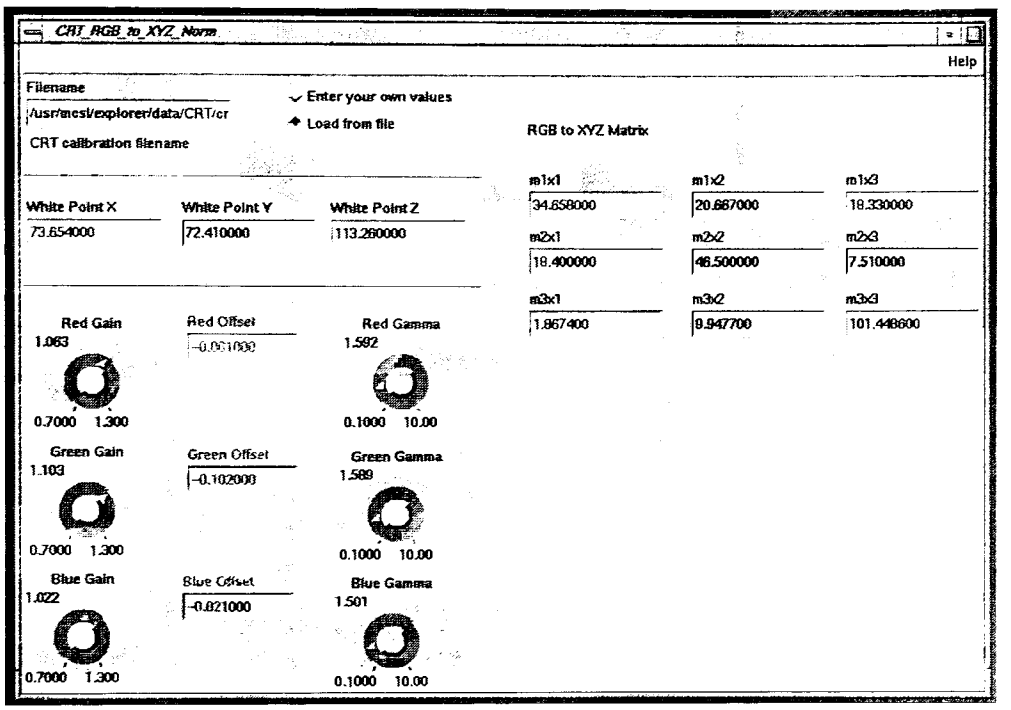
[13] *FlashPix Version 1.0 Format Specification*, Eastman Kodak Company, Rochester, 1996.

[14] *C. Hauf, E. Giorgianni, K. Spaulding, and S. Gregory, "Kodak PhotoYCC Color Interchange Space:Definitions and Relations",* Version 1.2, White paper, Eastman Kodak Company, Rochester (1996).

[15] M. Sebern, *Building OSF/Motif Applications: A Practical Introduction,* Prentice Hall, New Jersey, 1994.

7. Appendix A - Help pages for Explorer modules

7.1 Device models



NAME

CRT_RGB_to_XYZ_Norm

DESCRIPTION

Module calculates the Berns et. al CRT gain, offset, gamma CRT model based upon optimized model parameters. Data can be read from a file in a specific format or can be entered by hand. Reference sample CRT data file given in /usr/mcsl/explorer/data/CRT.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, byte, uniform). Input image or data sets of 8 bit per channel CRT RGB video code values

WIDGETS

Filename -- Text

Filename (use complete filename path) for the CRT data

file. Only required if data is to be brought in to the module through the use file option.

Which -- Radio Box

(Enter your own values, Load from file). Sets up module to allow user to enter values from a file or from the keyboard directly into the widgets.

NOTE: If data is loaded from a file, but then a user wishes to change the parameters interactively, this widget must be switched to "Enter your own values". All of the values will remain, however, the user can now interact and change them.

White Point X -- Text

Measured X tristimulus value of the CRT. This value will be used to normalize the resulting CIE XYZ tristimulus values from the model.

White Point Y -- Text

Measured Y tristimulus value of the CRT. This value will be used to normalize the resulting CIE XYZ tristimulus values from the model.

White Point Z -- Text

Measured Z tristimulus value of the CRT. This value will be used to normalize the resulting CIE XYZ tristimulus values from the model.

Red Gain -- Dial

Widget used to modify the gain term in the model for the red channel.

Red Offset -- Text

Widget used to display the resulting offset term resulting from a change in the gain term. Since the two values must equal 1.0, the offset term is not available for adjustment by the user. To adjust the offset, the gain term must be adjusted.

Red Gamma -- Dial

Widget used to modify the gamma term in the model for the red channel.

Green Gain -- Dial

Widget used to modify the gain term in the model for the green channel.

Green Offset -- Text

Green Gamma -- Dial

Widget used to modify the gamma term in the model for the green channel.

Blue Gain -- Dial

Widget used to modify the gain term in the model for the blue channel.

Blue Offset -- Text
Widget used to display the resulting offset term resulting from a change in the gain term. Since the two values must equal 1.0, the offset term is not available for adjustment by the user. To adjust the offset, the gain term must be adjusted.

Blue Gamma -- Dial
Widget used to modify the gamma term in the model for the blue channel.

m1x1 -- Text
Row 1 Column 1 value of the RGB to XYZ primary transformation matrix.

m1x2 -- Text
Row 1 Column 2 value of the RGB to XYZ primary transformation matrix.

m1x3 -- Text
Row 1 Column 3 value of the RGB to XYZ primary transformation matrix.

m2x1 -- Text
Row 2 Column 1 value of the RGB to XYZ primary transformation matrix.

m2x2 -- Text
Row 2 Column 2 value of the RGB to XYZ primary transformation matrix.

m2x3 -- Text
Row 2 Column 3 value of the RGB to XYZ primary transformation matrix.

m3x1 -- Text
Row 3 Column 1 value of the RGB to XYZ primary transformation matrix.

m3x2 -- Text
Row 3 Column 2 value of the RGB to XYZ primary transformation matrix.

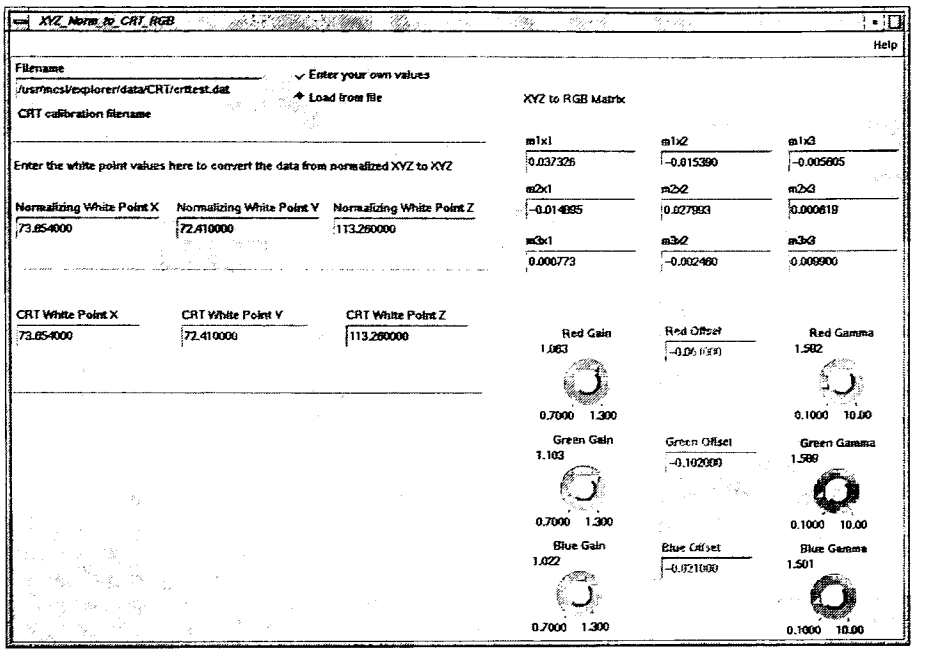
m3x3 -- Text
Row 3 Column 3 value of the RGB to XYZ primary transformation matrix.

OUTPUTS

Output -- Lattice
(1..2-D, 1..3-vector, double, uniform). Output image or data set in normalized CIE 1931 XYZ tristimulus values. Normalizing white point is chosen to be the CRT

white point.

KNOWN PROBLEMS
SEE ALSO



NAME

XYZ_Norm_to_CRT_RGB

DESCRIPTION

Module calculates the Berns et. al CRT gain, offset, gamma CRT model based upon optimized model parameters. Data can be read from a file in a specific format or can be entered by hand. Reference sample CRT data file given in /usr/mcsl/explorer/data/CRT.

Copyright 1997

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set in normalized CIE 1931 tristimulus values

WIDGETS

Filename -- Text

Filename (use complete filename path) for the CRT data file. Only required if data is to be brought in to the module through the use file option.

Which -- Radio Box

(Enter your own values, Load from file). Sets up module to allow user to enter values from a file or from the keyboard directly into the widgets.

NOTE: If data is loaded from a file, but then a user

wishes to change the parameters interactively, this widget must be switched to "Enter your own values". All of the values will remain, however, the user can now interact and change them.

CRT White Point X -- Text
Measured X tristimulus value of the CRT.

CRT White Point Y -- Text
Measured Y tristimulus value of the CRT.

CRT White Point Z -- Text
Measured Z tristimulus value of the CRT.

Red Gain -- Dial
Widget used to modify the gain term in the model for the red channel.

Red Offset -- Text
Widget used to display the resulting offset term resulting from a change in the gain term. Since the two values must equal 1.0, the offset term is not available for adjustment by the user. To adjust the offset, the gain term must be adjusted.

Red Gamma -- Dial
Widget used to modify the gamma term in the model for the red channel.

Green Gain -- Dial
Widget used to modify the gain term in the model for the green channel.

Green Offset -- Text
Widget used to display the resulting offset term resulting from a change in the gain term. Since the two values must equal 1.0, the offset term is not available for adjustment by the user. To adjust the offset, the gain term must be adjusted.

Green Gamma -- Dial
Widget used to modify the gamma term in the model for the green channel.

Blue Gain -- Dial
Widget used to modify the gain term in the model for the blue channel.

Blue Offset -- Text
Widget used to display the resulting offset term resulting from a change in the gain term. Since the two values must equal 1.0, the offset term is not available for adjustment by the user. To adjust the offset, the gain term must be adjusted.

Blue Gamma -- Dial
Widget used to modify the gamma term in the model for the blue channel.

m1x1 -- Text
Row 1 Column 1 value of the XYZ to RGB primary transformation matrix.

m1x2 -- Text
Row 1 Column 2 value of the XYZ to RGB primary transformation matrix.

m1x3 -- Text
Row 1 Column 3 value of the XYZ to RGB primary transformation matrix.

m2x1 -- Text
Row 2 Column 1 value of the XYZ to RGB primary transformation matrix.

m2x2 -- Text
Row 2 Column 2 value of the XYZ to RGB primary transformation matrix.

m2x3 -- Text
Row 2 Column 3 value of the XYZ to RGB primary transformation matrix.

m3x1 -- Text
Row 3 Column 1 value of the XYZ to RGB primary transformation matrix.

m3x2 -- Text
Row 3 Column 2 value of the XYZ to RGB primary transformation matrix.

m3x3 -- Text
Row 3 Column 3 value of the XYZ to RGB primary transformation matrix.

Normalizing White Point X -- Text
CIE 1931 X tristimulus value used to remove the normalization in the input data for processing through the model.

Normalizing White Point Y -- Text
CIE 1931 Y tristimulus value used to remove the normalization in the input data for processing through the model.

Normalizing White Point Z -- Text
CIE 1931 Y tristimulus value used to remove the normalization in the input data for processing through the model.

OUTPUTS

Output -- Lattice

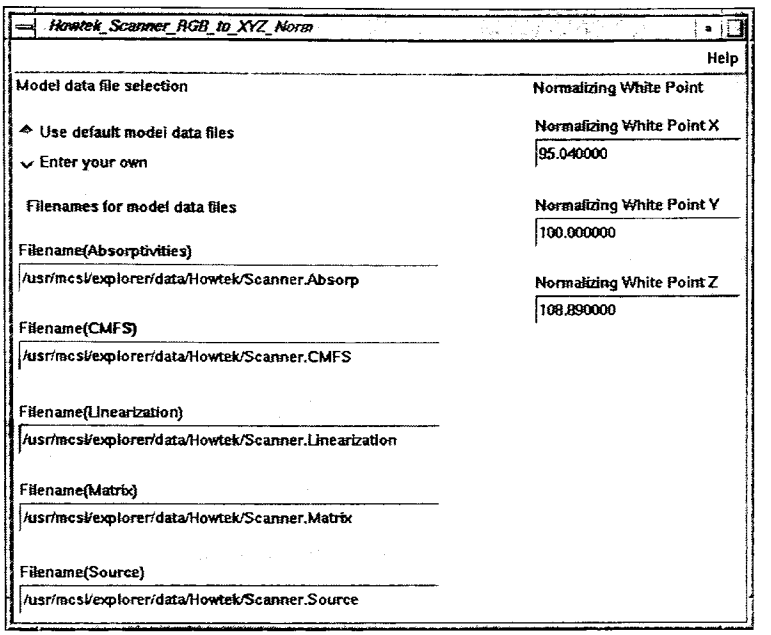
(1..2-D, 1..3-vector, byte, uniform). Output image or data set of 8 bit per channel, RGB code values resulting from the CRT model

KNOWN PROBLEMS

Accepts only double data on input

SEE ALSO

CRT_RGB_to_XYZ_Norm



NAME

Howtek_Scanner_RGB_to_XYZ_Norm

DESCRIPTION

Module to apply the Berns et. al spectral scanner model. This module works with the Munsell Color Science Labs Howtek scanner, however, it can be applied generally to any scanner modelled using this methodology.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, byte, uniform). Input image or data set in 8 bit per channel scanner RGB code values.

WIDGETS

Normalizing White Point X -- Text

X CIE tristimulus value of a white point to be used to normalize the data.

Normalizing White Point Y -- Text

Y CIE tristimulus value of a white point to be used to normalize the data.

Normalizing White Point Z -- Text

Z CIE tristimulus value of a white point to be used to normalize the data.

Filename(Absorptivities) -- Text

Filename (use complete path) for absortivities ascii

data file.

Filename(CMFS) -- Text
Filename (use complete path) for color matching functions(CMFS) ascii data file.

Filename(Linearization) -- Text
Filename (use complete path) for the linearization ascii data file.

Filename(Matrix) -- Text
Filename (use complete path) for matrix ascii data file.

Filename(Source) -- Text
Filename (use complete path) for the source data ascii data file.

Use default data files -- Radio Box
(Use default model data files, Enter your own). Allows user to choose between default model files used as an example for the Howtek scanner or to enter their own to generalize the use of this model.

OUTPUTS

Output -- Lattice
(1..2-D, 1..3-vector, double, uniform). Output image or data set in normalized CIE XYZ tristimulus values.

KNOWN PROBLEMS

Requires source code change and recompilation for a different set of default files.

SEE ALSO

NAME

Solitaire_RGB_to_XYZ_Norm

DESCRIPTION

Module to apply Berns et. al. film recorder model. This module is customized for the Munsell Labs MGI Solitaire film recorder, but can also be used for other film recorders by inputting the proper model data files.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, byte, uniform). Input image or data set in 8 bit film recorder RGB values.

WIDGETS

Normalizing White Point X -- Text

X tristimulus value for the white point which will be used to normalize the resulting XYZ tristimulus values.

Normalizing White Point Y -- Text

Y tristimulus value for the white point which will be used to normalize the resulting XYZ tristimulus values.

Normalizing White Point Z -- Text

Z tristimulus value for the white point which will be used to normalize the resulting XYZ tristimulus values.

Filename(Luts) -- Text

Filename (use complete pathname) for the luts used in the model calculation.

Filename(Weights) -- Text

Filename (use complete pathname) for the tristimulus weights used in the model calculation.

Filename(Vectors) -- Text

Filename (use complete pathname) for the vectors used in the model calculation.

Choice -- Radio Box

(Use default model data files, Enter your own). Allows user to choose between default sample model files for the Solitaire film recorder or enter their own for another device or model setup.

OUTPUTS

Output -- Lattice

(1..2-D, 1..3-vector, double, uniform). Output image

or data set in normalized CIE tristimulus values.

KNOWN PROBLEMS

SEE ALSO

XYZ_Norm_to_Solitaire_RGB

NAME

XYZ_Norm_to_Solitaire_RGB

DESCRIPTION

Module to apply Berns et. al. film recorder model. This module is customized for the Munsell Labs MGI Solitaire film recorder, but can also be used for other film recorders by inputting the proper model data files.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set in normalized XYX tristimulus values.

WIDGETS

Normalizing White Point X -- Text

X tristimulus value for the white point which will be used to remove the white point normalization of the input XYZ data.

Normalizing White Point Y -- Text

Y tristimulus value for the white point which will be used to remove the white point normalization of the input XYZ data.

Normalizing White Point Z -- Text

Z tristimulus value for the white point which will be used to remove the white point normalization of the input XYZ data.

Filename(Luts) -- Text

Filename (use complete pathname) for the luts used in the model calculation.

Filename(Weights) -- Text

Filename (use complete pathname) for the tristimulus weights used in the model calculation.

Filename(Vectors) -- Text

Filename (use complete pathname) for the vectors used in the model calculation.

Choice -- Radio Box

(Use default model data files, Enter your own). Allows user to choose between default sample model files for the Solitaire film recorder or enter their own for another device or model setup.

OUTPUTS

Output -- Lattice

(1..2-D, 1..3-vector, byte, uniform). Output image or data set in Solitaire film recorder 8 bit per channel RGB code values.

KNOWN PROBLEMS

Module uses "brute force" iteration method which can have convergence problems.

Accepts only double data on input.

SEE ALSO

Solitaire_RGB_to_XYZ_Norm

7.2 Color appearance models

NAME

XYZ_Norm_to_Hunt93

DESCRIPTION

Module to calculate the Hunt93 color appearance parameters, JCh, from normalized CIE 1931 tristimulus values. This module can be used to calculate the full model or can be used to pre-calculate the input Hunt model parameters for use when inverting the model.

INPUTS

Input -- Lattice

(2-D, 1..3-vector, double, uniform). Input image or data set in normalized CIE tristimulus values

WIDGETS

White Point X -- Text

X CIE tristimulus value for white point for Hunt model calculation

White Point Y -- Text

Y CIE tristimulus value for white point for Hunt model calculation

White Point Z -- Text

Z CIE tristimulus value for white point for Hunt model calculation

N White Point X -- Text

X CIE tristimulus value for the white point used to

normalize the input data. This parameter is required to remove the normalization for proper calculation of the Hunt93 which needs non-normalized CIE XYZ values.

N White Point Y -- Text
Y CIE tristimulus value for the white point used to normalize the input data. This parameter is required to remove the normalization for proper calculation of the Hunt93 which needs non-normalized CIE XYZ values.

N White Point Z -- Text
Z CIE tristimulus value for the white point used to normalize the input data. This parameter is required to remove the normalization for proper calculation of the Hunt93 which needs non-normalized CIE XYZ values.

White Luminance -- Text
Value for the white luminance needed to calculate the Hunt93 model parameters.

Approx CCT of Source -- Text
Value for the approximate correlated color temperature (CCT) of the source to be used in calculating the Hunt93 model.

Bk Rel Per Luminance -- Text
Value for the black relative percent luminance value used in calculating the Hunt93 model.

Choice of surround -- Radio Box
(Small areas in uniform light backgrounds and surrounds, Normal scenes, Television and CRT displays in dim surrounds, Projected photographs in dark surrounds, Arrays of adjacent colors in dark surrounds, Large transparencies on viewers). Choice of surround type based on choices given by Hunt. Different choices select different parameters for input into the Hunt93 model calculation.

Softcopy or Hardcopy -- Radio Box
(Softcopy, Hardcopy). Choice of whether calculations are being made for hardopy media like prints or softcopy media like CRT displays.

Precalc only -- Radio Box
(Precalc only, Run full model). Widget to choose whether to run the precalculation and the full model calculation or just the precalculation step.

OUTPUTS
Output -- Lattice
(2-D, 1..3-vector, double, uniform). Output image or data set in the Hunt93 JCh coodinates.

Model Data -- Lattice

(1-D, 1-vector, double, uniform). Lattice of the Hunt93 input model parameters resulting from the precalculation mode.

KNOWN PROBLEMS

Accepts only double data

SEE ALSO

Hunt93_to_XYZ_Norm

NAME

Hunt93_to_XYZ_Norm

DESCRIPTION

Module to calculate the inverse of the Hunt93 color appearance model. This module uses an iterative technique since no inverse set of model equations exists for the Hunt model.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input Image -- Lattice

(2-D. 1..3-vector, double. uniform). Input image or data set in the Hunt93 JCh parameters.

Model Data -- Lattice

(1-D. 1-vector, double. uniform). Input lattice of pre-calculated Hunt93 model data for use in iteratively calculating the inverse. This data is available by using the Pre-Calc function of the XYZ_Norm_to_Hunt93 module.

WIDGETS

Renorm White Point X -- Text

X tristimulus value of the white point to be used for normalization.

Renorm White Point Y -- Text

Y tristimulus value of the white point to be used for normalization.

Renorm White Point Z -- Text

Z tristimulus value of the white point to be used for normalization.

OUTPUTS

Output -- Lattice

(1..2-D, 1..3-vector, double. uniform). Output image or data set in normalized CIE XYZ tristimulus values.

KNOWN PROBLEMS

Performance depends on ability of iterative technique to converge.

Only accepts double data.

SEE ALSO

XYZ_Norm_to_Hunt93

NAME

XYZ_Norm_to_RLAB

DESCRIPTION

Module to apply the RLAB color appearance model to calculate the RLAB Lr, ar, br model parameters.

Christoper R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set in normalized CIE XYZ tristimulus values.

WIDGETS

White Point X -- Text

X tristimulus value of the white point to be used to unnormalize the input normalized XYZ tristimulus values for input into the RLAB model.

White Point Y -- Text

Y tristimulus value of the white point to be used to unnormalize the input normalized XYZ tristimulus values for input into the RLAB model.

White Point Z -- Text

Z tristimulus value of the white point to be used to unnormalize the input normalized XYZ tristimulus values for input into the RLAB model.

Adapting WP X -- Text

X CIE tristimulus value of the adapting white point used in the RLAB model

Adapting WP Y -- Text

Y CIE tristimulus value of the adapting white point used in the RLAB model

Adapting WP Z -- Text

Z CIE tristimulus value of the adapting white point used in the RLAB model

Adapting Luminance -- Text

Value of the adapting luminance used in the RLAB model calculation.

Discounting Illuminant Factor -- Slider

Value of the discounting illuminant factore used in the RLAB model calculation.

Sigma -- Radio Box
(Average (1 / 2.3), Dim (1 / 2.9), Dark (1 / 3.5), Custom (1/value)). Choice of sigma to be applied in the RLAB calculation.

Custom sigma -- Dial
Allows for the input of a custom sigma value.

OUTPUTS

Output -- Lattice
(1..2-D, 1..3-vector, double, uniform). Output image or data set in the RLAB Lr, ar, br parameters.

KNOWN PROBLEMS

Accepts only double data.

SEE ALSO

RLAB_to_XYZ_Norm

NAME

RLAB_to_XYZ_Norm

DESCRIPTION

Module to apply the inverse RLAB color appearance model to calculate normalized XYZ tristimulus values from the RLAB coordinates.

Christoper R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set in the RLAB coordinates, Lr, ar, br

WIDGETS

White Point X -- Text

X tristimulus value for the white point which will be used to normalize the resulting XYZ tristimulus values.

White Point Y -- Text

Y tristimulus value for the white point which will be used to normalize the resulting XYZ tristimulus values.

White Point Z -- Text

Z tristimulus value for the white point which will be used to normalize the resulting XYZ tristimulus values.

Adapting WP X -- Text

X CIE tristimulus value of the adapting white point used in the RLAB model

Adapting WP Y -- Text

Y CIE tristimulus value of the adapting white point used in the RLAB modelY CIE tristimulus value of the adapting white point used in the RLAB model

Adapting WP Z -- Text

Z CIE tristimulus value of the adapting white point used in the RLAB model

Adapting Luminance -- Text

Value of the adapting luminance used in the RLAB model calculation.

Discounting Illuminant Factor -- Slider

Value of the discounting illuminant factore used in the RLAB model calculation.Value of the discounting illuminant factore used in the RLAB model calculation.

Sigma -- Radio Box
(Average (1 / 2.3), Dim (1 / 2.9), Dark (1 / 3.5), Custom (1/value)). Choice of sigma to be applied in the RLAB calculation.

Custom sigma -- Dial
Allows for the input of a custom sigma value.

OUTPUTS

Output -- Lattice
(1..2-D, 1..3-vector, double, uniform). Output image or data set in normalizing CIE XYZ tristimulus values

KNOWN PROBLEMS

Accepts only double data on input

SEE ALSO

XYZ_Norm_to_RLAB

NAME

XYZ_Norm_to_RLAB_Variable

DESCRIPTION

Module to apply the RLAB color appearance model to create the RLAB Lr, ar, br output parameters. This module adds the additional capability for being able to apply variable sigmas on Lr and ar/br.

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set in normalized CIE XYZ tristimulus values.

WIDGETS

White Point X -- Text

X tristimulus value of the white point to be used to remove the normalization of the input data.

White Point Y -- Text

Y tristimulus value of the white point to be used to remove the normalization of the input data.

White Point Z -- Text

Z tristimulus value of the white point to be used to remove the normalization of the input data.

Adapting WP X -- Text

X CIE tristimulus value of the adapting white point used in the RLAB model

Adapting WP Y -- Text

Y CIE tristimulus value of the adapting white point used in the RLAB model

Adapting WP Z -- Text

Z CIE tristimulus value of the adapting white point used in the RLAB model

Adapting Luminance -- Text

Value of the adapting luminance used in the RLAB model calculation.

Discounting Illuminant Factor -- Slider

Value of the discounting illuminant factore used in the RLAB model calculation.

Sigma Lr/all -- Radio Box

(Average (1 / 2.3), Dim (1 / 2.9), Dark (1 / 3.5), Custom (1/value)). Choice of the sigma for all of the channels or just the Lr channel if the variable sigma widget has been chosen.

Custom sigma Lr/all -- Dial

Allows for the input of a custom sigma value.

Sigma ar/br -- Radio Box
(Average (1 / 2.3), Dim (1 / 2.9), Dark (1 / 3.5), Custom (1 / value)). Choice of sigma for ar/br if the variable sigma widget has been set to variable.

Custom sigma ar/br -- Dial
Allows for the input of a custom sigma value for ar/br.

Variable exponents -- Radio Box
(Same sigma all three channels Lr, ar, br, Variable sigma Lr & ar,br). Allows for the choice of the standard method of calculating RLAB where the same sigma is applied to all three channels or the variable method where a different sigma can be applied to Lr and ar/br respectively.

OUTPUTS

Output -- Lattice
(1..2-D, 1..3-vector, double, uniform). Output image or dats set in the RLAB coordinates, Lr, ar, br.

KNOWN PROBLEMS

Accepts only double data.

SEE ALSO

RLAB_to_XYZ_Norm_Variable

NAME

RLAB_to_XYZ_Norm_Variable

DESCRIPTION

Module to apply the inverse RLAB color appearance model to create output CIE normalized XYZ tristimulus values. This module adds the additional capability for being able to deal with variable sigmas on Lr and ar/br.

Christoper R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set in the RLAB coordinates, Lr, ar, br.

WIDGETS

White Point X -- Text

X tristimulus value of the white point to be used to normalize the resulting XYZ tristimulus values of the RLAB model for output.

White Point Y -- Text

Y tristimulus value of the white point to be used to normalize the resulting XYZ tristimulus values of the RLAB model for output.

White Point Z -- Text

Z tristimulus value of the white point to be used to normalize the resulting XYZ tristimulus values of the RLAB model for output.

Adapting WP X -- Text

X CIE tristimulus value of the adapting white point used in the RLAB model calculation.

Adapting WP Y -- Text

Y CIE tristimulus value of the adapting white point used in the RLAB model calculation.

Adapting WP Z -- Text

Z CIE tristimulus value of the adapting white point used in the RLAB model calculation.

Adapting Luminance -- Text

Value of the adapting luminance used in the RLAB model calculation.

Discounting Illuminant Factor -- Slider

Value of the discounting illuminant factore used in the

RLAB model calculation.

Sigma Lr/all -- Radio Box
(Average (1 / 2.3), Dim (1 / 2.9), Dark (1 / 3.5), Custom (1/value)). Choice of the sigma for all of the channels or just the Lr channel if the variable sigma widget has been chosen.

Custom sigma Lr/all -- Dial
Allows for the input of a custom sigma value.

Sigma ar/br -- Radio Box
(Average (1 / 2.3), Dim (1 / 2.9), Dark (1 / 3.5), Custon (1 / value)). Choice of sigma for ar/br if the variable sigma widget has been set to variable.

Custom sigma ar/br -- Dial
Allows for the input of a custom sigma value for ar/br.

Variable exponents -- Radio Box
(Same sigma all three channels Lr, ar, br, Variable sigma Lr & ar,br). Allows for the choice of the standard method of calculating RLAB where the same sigma is applied to all three channels or the variable method where a different sigma can be applied to Lr and ar/br respectively.

OUTPUTS
Output -- Lattice
(1..2-D, 1..3-vector, double, uniform). Output image or data set in normalized XYZ tristimulus values.

KNOWN PROBLEMS
Accepts only double data on input.

SEE ALSO
XYZ_Norm_to_RLAB_Variable

7.3 Color space transformations

NAME
XYZ_Norm_to_CIELab

DESCRIPTION
Module accepts normalized CIE 1931 XYZ tristimulus values and converts those values to CIE 1976 CIELab values as output. The Xn, Yn, Zn used in the CIELab calculation are assumed to be equal to the normalizing white point of the incoming data.

INPUTS
Input -- Lattice
(1..2-D, 1..3-vector, double, uniform).

Input image or data are in normalized CIE 1931 XYZ tristimulus values.

OUTPUTS
Output -- Lattice
(1..2-D, 1..3-vector, double, uniform).

Output image or data are in 1976 CIELab values

KNOWN PROBLEMS
Module only accepts double data.

Data must be normalized to a white point prior to entering this module. Non-normalized data will be improperly processed.

SEE ALSO
CIELab_to_XYZ_Norm

NAME

CIELab_to_XYZ_Norm

DESCRIPTION

Module accepts values in 1976 CIELab values and converts those values to normalized CIE 1931 tristimulus values. The normalizing white point is assumed to be the Xn,Yn,Zn used in the calculation of the CIELab values.

Christopher R. Hauf

Munsell Color Science Laboratory Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform).

Input image or data is in 1976 CIELab values

OUTPUTS

Output -- Lattice

(1..2-D, 1..3-vector, double, uniform).

Output image is in normalized CIE 1931 tristimulus values.

KNOWN PROBLEMS

Module only accepts double data

Data exiting the module is still normalized to whatever white point was used in the calculation of the original CIELab values.

SEE ALSO

XYZ_Norm_to_CIELab

NAME
CIELab_to_CIELCh

DESCRIPTION
Module to convert 1976 CIELab values to 1976 CIELCh values

Copyright 1997

Christopher R. Hauf

Munsell Color Science Laboratory Rochester Institute of Technology

INPUTS
Input -- Lattice
(1..2-D, 1..3-vector, double, uniform).

Input image or data in 1976 CIELab values

OUTPUTS
Output -- Lattice
(1..2-D, 1..3-vector, double, uniform).

Output image or data in 1976 CIELCh values

KNOWN PROBLEMS
Module only accepts double data on input

SEE ALSO
CIELCh_to_CIELab

NAME

CIELCh_to_CIELab

DESCRIPTION

Module to convert CIELCh coordinates to 1976 CIE CIELab coordinates

Christopher R. Hauf

Munsell Color Science Laboratory Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform).

Input image or data in CIELCh coordinates.

OUTPUTS

Output -- Lattice

(1..2-D, 1..3-vector, double, uniform).

Output image or data in CIELab coordinates.

KNOWN PROBLEMS

Module only accepts double data on input.

SEE ALSO

CIELab_to_CIELCh

NAME

XYZ_Norm_to_CIELuv

DESCRIPTION

Module calculates CIE 1976 CIELUV coordinates from normalized CIE XYZ tristimulus values.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set in normalized CIE XYZ tristimulus values.

WIDGETS

White Point X -- Text

Input X tristimulus value for the normalizing white point which will be used to remove the normalization for the CIELUV calculation.

White Point Y -- Text

Input Y tristimulus value for the normalizing white point which will be used to remove the normalization for the CIELUV calculation.

White Point Z -- Text

Input Z tristimulus value for the normalizing white point which will be used to remove the normalization for the CIELUV calculation.

OUTPUTS

Output -- Lattice

(1..2-D, 1..3-vector, double, uniform). Output image or data set in 1976 CIELUV coordinates.

u'n -- Parameter

u' chromaticity coordinate for the white point.

v'n -- Parameter

v' chromaticity coodinate for the white point.

KNOWN PROBLEMS

Accepts only double data.

SEE ALSO

NAME

sRGB_to_Norm_XYZ

DESCRIPTION

Module to calculate normalized CIE 1931 XYZ tristimulus values for Stokes et al. sRGB 8bit per channel RGB code values.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(2-D, 1..3-vector, byte, uniform). Input image or data set in sRGB 8 bit code values

WIDGETS

White Point X -- Text

X 1931 CIE tristimulus value used to normalize resulting XYZ values for output.

White Point Y -- Text

Y 1931 CIE tristimulus value used to normalize resulting XYZ values for output.

White Point Z -- Text

Z 1931 CIE tristimulus value used to normalize resulting XYZ values for output.

OUTPUTS

Output -- Lattice

(2-D, 1..3-vector, double, uniform). Output image or data set in CIE 1931 normalized tristimulus values.

KNOWN PROBLEMS

<Describe any known problems/limitations here>

SEE ALSO

<List associated or related modules here>

NAME

PhotoYCC_to_Norm_XYZ

DESCRIPTION

Module to calculate normalized CIE 1931 XYZ tristimulus from Kodak PhotoYCC Color Interchange Space code values.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input Image -- Lattice

(2-D, 1..3-vector, byte, uniform). Input image or data in the 8bit/channel Kodak PhotoYCC values.

WIDGETS

White Point X -- Text

CIE X tristimulus value for the white point to be used to normalize the data.

White Point Y -- Text

CIE Y tristimulus value for the white point to be used to normalize the data.

White Point Z -- Text

CIE Z tristimulus value for the white point to be used to normalize the data.

OUTPUTS

Output Image -- Lattice

(2-D, 1..3-vector, double, uniform). Image or data set in normalized CIE 1931 XYZ tristimulus values

KNOWN PROBLEMS

SEE ALSO

7.4 Mathematical operations

NAME
Channel_Gain_and_Offset

DESCRIPTION
Module to apply a channel independent gain and offset to the incoming data.

Copyright 1997

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS
Input -- Lattice
(1..2-D, 1..3-vector, double, uniform). Input image or data set to have the channel independent gain and offsets applied

WIDGETS
Red gain -- Text
Value of the red gain to be multplied with the incoming channel 1 value. Default value is 1.0.

NOTE: The term red is used here to indicate channel 1. Therefore, the module will work on other types of data.

Red offset -- Text
Value of the channel 1 offset term which will be added to the channel 1 value. Default value is 0.0.

Green gain -- Text
Channel 2 gain term to be multiplied with the incoming channel 2 value. Default value is 1.0.

Green offset -- Text
Value of the channel 2 offset term which will be added to the channel 2 value. Default value is 0.0.

Blue gain -- Text
Channel 3 gain term to be multiplied with the incoming channel 3 value. Default value is 1.0.

Blue offset -- Text
Value of the channel 3 offset term which will be added to the channel 3 value. Default value is 0.0.

OUTPUTS
Output -- Lattice
(1..2-D, 1..3-vector, double, uniform). Output image or data set with the values applied.

KNOWN PROBLEMS

Only accepts double data on input.

SEE ALSO

NAME

Channel_Power

DESCRIPTION

Module used to apply a channel independent exponent to the incoming data.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set to have exponent applied.

WIDGETS

Channel 1 Exponent Value -- Dial

Value of the channel 1 exponent. Default value is 1.0.

Channel 2 Exponent Value -- Dial

Value of the channel 2 exponent. Default value is 1.0.

Channel 3 Exponent Value -- Dial

Value of the channel 3 exponent. Default value is 1.0.

OUTPUTS

Output -- Lattice

(1..2-D, 1..3-vector, double, uniform). Output image or data set of values with exponent applied.

KNOWN PROBLEMS

Only accepts double data as input.

SEE ALSO

NAME

Sigmoid

DESCRIPTION

Module to apply a channel independent sigmoid like value to the input data.

Copyright 1997

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(1..2-D, 1..3-vector, double, uniform). Input image or data set to have sigmoid function applied to it.

WIDGETS

Channel 1 Sigmoid Value -- Dial

Adjusts curvature of the sigmoid like curve for channel 1.

Channel 2 Sigmoid Value -- Dial

Adjusts curvature of the sigmoid like curve for channel 2.

Channel 3 Sigmoid Value -- Dial

Adjusts curvature of the sigmoid like curve for channel 3.

OUTPUTS

Output -- Lattice

(1..2-D, 1..3-vector, double, uniform). Output image or data set with sigmoid applied.

KNOWN PROBLEMS

Accepts only double data on input.

SEE ALSO

NAME

3X3_Matrix_Multiply

DESCRIPTION

Module to perform a 3x3 matrix multiplication with the 3 component input data. Values for the matrix can be entered by hand or read out of a space/tab delimited ASCII text file.

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(2-D, 1..3-vector, double, uniform). Input image or data set to be multiplied by the 3x3 matrix.

WIDGETS

Which -- Radio Box

(Enter Values, Load from file). Allows choice of whether to load values from file or from the keyboard.

NOTE: If values are loaded from a file and then wished to be modified through the keyboard, this widget needs to be switched from "Load from File" to "Enter Values". All of the values will remain unchanged when the switch is made, but will now allow for them to be interactively changed.

Filename -- Text

Filename (use full pathname) of the ASCII file containing the 3x3 matrix coefficients.

m1x1 -- Text

Row 1 Column 1 matrix value.

m1x2 -- Text

Row 1 Column 2 matrix value.

m1x3 -- Text

Row 1 Column 3 matrix value.

m2x1 -- Text

Row 2 Column 1 matrix value.

m2x2 -- Text

Row 2 Column 2 matrix value.

m2x3 -- Text

Row 2 Column 3 matrix value.

m3x1 -- Text
Row 3 Column 1 matrix value.

m3x2 -- Text
Row 3 Column 2 matrix value.

m3x3 -- Text
Row 3 Column 3 matrix value.

OUTPUTS

Output -- Lattice
(2-D, 1..3-vector, double, uniform). Output image or data set which has been multiplied through the 3x3 matrix.

Input_2 -- Lattice
(2-D, 1..3-vector, uniform). Output image or data set which is an exact, unmodified copy of the original.

KNOWN PROBLEMS

Allows only double data on input

SEE ALSO

7.5 Look-up tables

NAME

3_1D_LUTS_byte

DESCRIPTION

Module allows for the application of channel independent 8bit 1 dimensionsal (1D) look-up tables (LUTS)

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input -- Lattice

(2..-D, 1..3-vector, byte, uniform). Input image or data set in 8bit per channel code values

WIDGETS

Filename LUT 1 -- Text

Filename (use complete path) of LUT file to be applied to channel 1. The LUT files are assumed to be ASCII text files of numbers with space or tab delimiting.

Filename LUT 2 -- Text

Filename (use complete path) of LUT file to be applied to channel 2. The LUT files are assumed to be ASCII text files of numbers with space or tab delimiting.

Filename LUT 3 -- Text

Filename (use complete path) of LUT file to be applied to channel 3. The LUT files are assumed to be ASCII text files of numbers with space or tab delimiting.

Look-up table (LUT) 1 -- Radio Box

(Don't use LUT, Use LUT). Allows for the choice of applying the LUT or not for channel 1.

Look-up table (LUT) 2 -- Radio Box

(Don't use LUT, Use LUT). Allows for the choice of applying the LUT or not for channel 2.

Look-up table (LUT) 3 -- Radio Box

(Don't use LUT, Use LUT). Allows for the choice of applying the LUT or not for channel 3.

OUTPUTS

Output -- Lattice

(2-D, 1..3-vector, byte, uniform). Output image or data set in 8bit per channel code values

KNOWN PROBLEMS

Only accepts LUTS with 256 entries.

SEE ALSO

<List associated or related modules here>

NAME

Read_3Component_Data

DESCRIPTION

<Replace this with your module description>

WIDGETS

Filename -- Text

<Describe the purpose of the widget here>

LUT Length -- Text

<Describe the purpose of the widget here>

Data Type -- Radio Box

(Unsigned char data, Double data). <Describe the purpose of the widget here>

OUTPUTS

Output LUT -- Lattice

(2-D, 3-vector, uniform). <Describe the purpose of the port here>

KNOWN PROBLEMS

<Describe any known problems/limitations here>

SEE ALSO

<List associated or related modules here>

7.6 Data management, support & analysis

NAME
Delta_E

DESCRIPTION
Module calculates an average delta E between two different input images or data sets and displays that delta E to the screen. The module also produces a single channel delta E image as output which can be used to see where in an image the largest error is occuring.

This module works with modules which output LAB data. Therefore, this module will work with the CIELAB module and the RLAB color appearance model module.

Copyright 1997

Christopher R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS
Image 1 -- Lattice
(2-D, 1..3-vector, double, uniform). First input image or data in Lab type coordinates for calculation.

Image 2 -- Lattice
(2-D, 1..3-vector, double, uniform). Second input image or data in Lab type coordinates for calculation.

WIDGETS
Average -- Text
Value shown is the average delta E calculated between the two data sets.

OUTPUTS
Delta E* image -- Lattice
(2-D, 1-vector, double, uniform). Single channel image which contains the pixelwise delta E values. Can be displayed or interogated to see where the largest error may be occuring within an image.

KNOWN PROBLEMS
The input images or data sets must be of the same size.

Only accepts double data on input

SEE ALSO
XYZ_Norm_to_CIELab
XYZ_Norm_to_RLAB
XYZ_Norm_to_RLAB_Variable

NAME

Select_White_Point

DESCRIPTION

Module to output the XYZ CIE tristimulus values for different CIE illuminants or a user specified illuminant. Useful for selecting normalizing white points.

WIDGETS

Which -- Radio Box

(D65, A, D50, D55, D75, C, Choose your own). Allows for the choice of standard CIE illuminants or to choose to enter a user defined value.

White Point X -- Text

X CIE tristimulus value for the white point.

White Point Y -- Text

Y CIE tristimulus value for the white point.

White Point Z -- Text

Z CIE tristimulus value for the white point.

OUTPUTS

Output White Point X -- Parameter

X CIE tristimulus value for the white point.

Output White Point Y -- Parameter

Y CIE tristimulus value for the white point.

Output White Point Z -- Parameter

Z CIE tristimulus value for the white point.

KNOWN PROBLEMS

SEE ALSO

NAME

Switch_Image

DESCRIPTION

Module to allow user switchable output of a single lattice from two different input lattices. Useful for comparing images in a single display window.

Christoper R. Hauf

Munsell Color Science Lab Rochester Institute of Technology

INPUTS

Input Image 1 -- Lattice

(2-D, 1..4-vector). First input image or data set. Allowance made for SGI RGBA with inclusion of the 4 data variables.

Input Image 2 -- Lattice

(2-D, 1..4-vector). Second input image or data set.

WIDGETS

Switch -- Radio Box

(Image 1, Image2). Choice of output lattice.

OUTPUTS

Output Image -- Lattice

(2-D, 1..4-vector). Output image or data set.

KNOWN PROBLEMS

SEE ALSO

8. ANSI C Source code

8.1 Device models

XYZ2crt.c *XYZ_Norm_to_CRT_RGB*

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


/*
        XYZ2crt.c

        This user function takes an input 2D image in normalized XYZ,
        renormalizes the data by the original white point and performs

        Chris Hauf
        March 1, 1995

        Repaired October 14, 1996 Chris Hauf


*/
void crt_to_XYZ_Calc(char *filename, int which, cxLattice *in,cxLattice **out,
                double white_X, double white_Y, double white_Z,
                double orig_white_X, double orig_white_Y, double orig_white_Z,
                double red_kg, double red_ko, double red_gamma,
                double green_kg, double green_ko, double green_gamma,
                double blue_kg, double blue_ko, double blue_gamma,
                double mat1x1, double mat1x2, double mat1x3,
                double mat2x1, double mat2x2, double mat2x3,
                double mat3x1, double mat3x2, double mat3x3
                )
{
        FILE *inputFile;
        int     i,j,k;          /* loop variables */
        unsigned char *b;       /* data pointers */
        double  *a;
        char text[80];
        cxErrorCode err;
        long dataLength;
        double rtemp, gtemp, btemp, temp1, temp2, temp3;
```

```
        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,               /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_byte);  /* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);


 switch(which)
 {



case 0:


  if(orig_white_X == NULL && orig_white_Y == NULL && orig_white_Z == NULL &&
    red_kg == NULL && red_ko == NULL && red_gamma == NULL &&
    green_kg == NULL && green_ko == NULL && green_gamma == NULL &&
    blue_kg == NULL && blue_ko == NULL && blue_gamma == NULL &&
    mat1x1 == NULL && mat1x2 == NULL && mat1x3 == NULL &&
    mat2x1 == NULL && mat2x2 == NULL && mat2x3 == NULL &&
    mat3x1 == NULL && mat3x2 == NULL && mat3x3 == NULL)
  {
        return;
  }
  else
  {



  cxInWdgtDblSet("Normalizing White Point X", orig_white_X);
  cxInWdgtDblSet("Normalizing White Point Y", orig_white_Y);
  cxInWdgtDblSet("Normalizing White Point Z", orig_white_Z);

  cxInWdgtDblMinMaxSet("Red Gain", 0.7, 1.3);
  cxInWdgtDblSet("Red Gain", red_kg);
  cxInWdgtEnable("Red Offset");
  red_ko = 1 - red_kg;
  cxInWdgtDblSet("Red Offset", red_ko);
  cxInWdgtDisable("Red Offset");
  cxInWdgtDblMinMaxSet("Red Gamma", 0.1, 10.0);
  cxInWdgtDblSet("Red Gamma", red_gamma);

  cxInWdgtDblMinMaxSet("Green Gain", 0.7, 1.3);
  cxInWdgtDblSet("Green Gain", green_kg);
  cxInWdgtEnable("Green Offset");
```

```
green_ko = 1 - green_kg;
cxInWdgtDblSet("Green Offset", green_ko);
cxInWdgtDisable("Green Offset");
cxInWdgtDblMinMaxSet("Green Gamma", 0.1, 10.0);
cxInWdgtDblSet("Green Gamma", green_gamma);

cxInWdgtDblMinMaxSet("Blue Gain", 0.7, 1.3);
cxInWdgtDblSet("Blue Gain", blue_kg);
cxInWdgtEnable("Blue Offset");
blue_ko = 1 - blue_kg;
cxInWdgtDblSet("Blue Offset", blue_ko);
cxInWdgtDisable("Blue Offset");
cxInWdgtDblSet("Blue Offset", blue_ko);
cxInWdgtDblMinMaxSet("Blue Gamma", 0.1, 10.0);
cxInWdgtDblSet("Blue Gamma", blue_gamma);

cxInWdgtDblSet("m1x1", mat1x1);
cxInWdgtDblSet("m1x2", mat1x2);
cxInWdgtDblSet("m1x3", mat1x3);
cxInWdgtDblSet("m2x1", mat2x1);
cxInWdgtDblSet("m2x2", mat2x2);
cxInWdgtDblSet("m2x3", mat2x3);
cxInWdgtDblSet("m3x1", mat3x1);
cxInWdgtDblSet("m3x2", mat3x2);
cxInWdgtDblSet("m3x3", mat3x3);

cxInWdgtDisable("CRT White Point X");
cxInWdgtDisable("CRT White Point Y");
cxInWdgtDisable("CRT White Point Z");

cxInWdgtHide("CRT White Point X");
cxInWdgtHide("CRT White Point Y");
cxInWdgtHide("CRT White Point Z");
}
```

```
/* loop over the data elements */
for ( i = 0; i < in->dims[1]; i++ )
{
        for ( j = 0; j < in->dims[0]; j++ )
        {


        rtemp = (((a[0] * orig_white_X) * mat1x1) +
                ((a[1] * orig_white_Y) * mat1x2) +
                ((a[2] * orig_white_Z) * mat1x3));

        gtemp = (((a[0] * orig_white_X) * mat2x1) +
                ((a[1] * orig_white_Y) * mat2x2) +
                ((a[2] * orig_white_Z) * mat2x3));

        btemp = (((a[0] * orig_white_X) * mat3x1) +
                ((a[1] * orig_white_Y) * mat3x2) +
                ((a[2] * orig_white_Z) * mat3x3));
```

```
if(rtemp >= 0.0 && rtemp <= 1.0)
{
        b[0]  = (unsigned char) rint((255.0 / red_kg) *
                ((pow(rtemp,  pow(red_gamma,  -1.0))) - red_ko));
}
else if(rtemp > 1.0)
{
        b[0] = 255;
}
else
{
        b[0] = 0;
}

if(gtemp >= 0.0 && gtemp <= 1.0)
{
        b[1] = (unsigned char) rint((255.0 / green_kg) *
                ((pow(gtemp, pow(green_gamma,  -1.0))) -
                green_ko));
}
else if(gtemp > 1.0)
{
        b[1] = 255;
}
else
{
        b[1] = 0;
}

if(btemp >= 0.0 && btemp <= 1.0)
{

        b[2] = (unsigned char) rint((255.0 / blue_kg) *
                ((pow(btemp, pow(blue_gamma,  -1.0))) -
                blue_ko));
}
else if(btemp > 1.0)
{
        b[2] = 255;
}
else
{
        b[2] = 0;
}



        a += 3;
        b += 3;


        }
```

```
break;


case 1:

if(filename == NULL) return;
if(filename[0] == NULL) return;



if((inputFile = fopen(filename, "r")) == NULL)
{
  cxModAlert("Can't open file!");
  return;
}



(void) fscanf(inputFile, "%s\n", text);


(void) fscanf(inputFile, "%lf %lf %lf",&white_X,
        &white_Y, &white_Z);


(void) fscanf(inputFile, "%s\n", text);


(void) fscanf(inputFile, "%lf  %lf  %lf", &red_ko,
        &red_kg, &red_gamma);
(void) fscanf(inputFile, "%lf %lf %lf", &green_ko,
        &green_kg, &green_gamma);
(void) fscanf(inputFile, "%lf %lf %lf", &blue_ko,
        &blue_kg, &blue_gamma);

(void) fscanf(inputFile, "%s\n", text);

(void) fscanf(inputFile, "%lf %lf %lf",&temp1, &temp2, &temp3);
(void) fscanf(inputFile, "%lf %lf %lf",&temp1, &temp2, &temp3);
(void) fscanf(inputFile, "%lf %lf %lf",&temp1, &temp2, &temp3);

(void) fscanf(inputFile, "%s\n", text);

(void) fscanf(inputFile, "%lf %lf %lf",&mat1x1, &mat1x2, &mat1x3);
(void) fscanf(inputFile, "%lf %lf %lf",&mat2x1, &mat2x2, &mat2x3);
(void) fscanf(inputFile, "%lf %lf %lf",&mat3x1, &mat3x2, &mat3x3);



fclose(inputFile);
```

```
cxInWdgtEnable("CRT White Point X");
cxInWdgtEnable("CRT White Point Y");
cxInWdgtEnable("CRT White Point Z");

cxInWdgtShow("CRT White Point X");
cxInWdgtShow("CRT White Point Y");
cxInWdgtShow("CRT White Point Z");


cxInWdgtDblSet("CRT White Point X", white_X);
cxInWdgtDblSet("CRT White Point Y", white_Y);
cxInWdgtDblSet("CRT White Point Z", white_Z);

cxInWdgtDblSet("Normalizing White Point X", orig_white_X);
cxInWdgtDblSet("Normalizing White Point Y", orig_white_Y);
cxInWdgtDblSet("Normalizing White Point Z", orig_white_Z);

cxInWdgtDblMinMaxSet("Red Gain", 0.7, 1.3);
cxInWdgtDblSet("Red Gain", red_kg);
cxInWdgtEnable("Red Offset");
cxInWdgtDblSet("Red Offset", red_ko);
cxInWdgtDisable("Red Offset");
cxInWdgtDblMinMaxSet("Red Gamma", 0.1, 10.0);
cxInWdgtDblSet("Red Gamma", red_gamma);

cxInWdgtDblMinMaxSet("Green Gain", 0.7, 1.3);
cxInWdgtDblSet("Green Gain", green_kg);
cxInWdgtEnable("Green Offset");
cxInWdgtDblSet("Green Offset", green_ko);
cxInWdgtDisable("Green Offset");
cxInWdgtDblMinMaxSet("Green Gamma", 0.1, 10.0);
cxInWdgtDblSet("Green Gamma", green_gamma);

cxInWdgtDblMinMaxSet("Blue Gain", 0.7, 1.3);
cxInWdgtDblSet("Blue Gain", blue_kg);
cxInWdgtEnable("Blue Offset");
cxInWdgtDblSet("Blue Offset", blue_ko);
cxInWdgtDisable("Blue Offset");
cxInWdgtDblSet("Blue Offset", blue_ko);
cxInWdgtDblMinMaxSet("Blue Gamma", 0.1, 10.0);
cxInWdgtDblSet("Blue Gamma", blue_gamma);

cxInWdgtDblSet("m1x1", mat1x1);
cxInWdgtDblSet("m1x2", mat1x2);
cxInWdgtDblSet("m1x3", mat1x3);
cxInWdgtDblSet("m2x1", mat2x1);
cxInWdgtDblSet("m2x2", mat2x2);
cxInWdgtDblSet("m2x3", mat2x3);
cxInWdgtDblSet("m3x1", mat3x1);
cxInWdgtDblSet("m3x2", mat3x2);
cxInWdgtDblSet("m3x3", mat3x3);
```

```
/* loop over the data elements */
for ( i = 0; i < in->dims[1]; i++ )
{
        for ( j = 0; j < in->dims[0]; j++ )
        {

        rtemp = (((a[0] * orig_white_X) * mat1x1) +
                ((a[1] * orig_white_Y) * mat1x2) +
                ((a[2] * orig_white_Z) * mat1x3));

        gtemp = (((a[0] * orig_white_X) * mat2x1) +
                ((a[1] * orig_white_Y) * mat2x2) +
                ((a[2] * orig_white_Z) * mat2x3));

        btemp = (((a[0] * orig_white_X) * mat3x1) +
                ((a[1] * orig_white_Y) * mat3x2) +
                ((a[2] * orig_white_Z) * mat3x3));


  if(rtemp >= 0.0 && rtemp <= 1.0)
  {
        b[0]  = (unsigned char) rint((255.0 / red_kg) *
                ((pow(rtemp,  pow(red_gamma,  -1.0))) - red_ko));
  }
  else if(rtemp > 1.0)
  {
        b[0] = 255;
  }
  else
  {
        b[0] = 0;
  }

  if(gtemp >= 0.0 && gtemp <= 1.0)
  {
        b[1] = (unsigned char) rint((255.0 / green_kg) *
                ((pow(gtemp, pow(green_gamma,  -1.0))) -
                green_ko));
  }
  else if(gtemp > 1.0)
  {
        b[1] = 255;
  }
  else
  {
        b[1] = 0;
  }

  if(btemp >= 0.0 && btemp <= 1.0)
  {

        b[2] = (unsigned char) rint((255.0 / blue_kg) *
                ((pow(btemp, pow(blue_gamma,  -1.0))) -
```

```
                        blue_ko));
                }
                else if(btemp > 1.0)
                {
                        b[2] = 255;
                }
                else
                {
                        b[2] = 0;
                }




                        a += 3;
                        b += 3;


                        }
        }

  break;




}

}
```

crt2XYZcalc.c ***CRT_RGB_to_XYZ_Norm***

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


/*
        crt2XYZcalc.c

        This user function takes an input 2D image in crt digital
        counts and creates an output image in XYZ.

        Chris Hauf
        March 1, 1995
```

```
*/
void crt_to_XYZ_Calc(char *filename, int which, cxLattice *in,cxLattice **out,
                double white_X, double white_Y, double white_Z,
                double red_kg, double red_ko, double red_gamma,
                double green_kg, double green_ko, double green_gamma,
                double blue_kg, double blue_ko, double blue_gamma,
                double mat1x1, double mat1x2, double mat1x3,
                double mat2x1, double mat2x2, double mat2x3,
                double mat3x1, double mat3x2, double mat3x3
                )
{
        FILE *inputFile;
        int     i,j,k;          /* loop variables */
        double *b;              /* data pointers */
        unsigned char *a;
        char text[80];
        cxErrorCode err;
        long dataLength;
        double rtemp, gtemp, btemp, temp;


        /* create the new lattice */
        *out = cxLatDataNew(
                2,              /* number of dimensions */
                in->dims,               /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);


 switch(which)
 {



case 0:


  if(white_X == NULL && white_Y == NULL && white_Z == NULL &&
    red_kg == NULL && red_ko == NULL && red_gamma == NULL &&
    green_kg == NULL && green_ko == NULL && green_gamma == NULL &&
    blue_kg == NULL && blue_ko == NULL && blue_gamma == NULL &&
    mat1x1 == NULL && mat1x2 == NULL && mat1x3 == NULL &&
    mat2x1 == NULL && mat2x2 == NULL && mat2x3 == NULL &&
    mat3x1 == NULL && mat3x2 == NULL && mat3x3 == NULL)
 {
        return;
 }
```

```
else
{

cxInWdgtDblSet("White Point X", white_X);
cxInWdgtDblSet("White Point Y", white_Y);
cxInWdgtDblSet("White Point Z", white_Z);

cxInWdgtDblMinMaxSet("Red Gain", 0.7, 1.3);
cxInWdgtDblSet("Red Gain", red_kg);
cxInWdgtEnable("Red Offset");
red_ko = 1 - red_kg;
cxInWdgtDblSet("Red Offset", red_ko);
cxInWdgtDisable("Red Offset");
cxInWdgtDblMinMaxSet("Red Gamma", 0.1, 10.0);
cxInWdgtDblSet("Red Gamma", red_gamma);

cxInWdgtDblMinMaxSet("Green Gain", 0.7, 1.3);
cxInWdgtDblSet("Green Gain", green_kg);
cxInWdgtEnable("Green Offset");
green_ko = 1 - green_kg;
cxInWdgtDblSet("Green Offset", green_ko);
cxInWdgtDisable("Green Offset");
cxInWdgtDblMinMaxSet("Green Gamma", 0.1, 10.0);
cxInWdgtDblSet("Green Gamma", green_gamma);

cxInWdgtDblMinMaxSet("Blue Gain", 0.7, 1.3);
cxInWdgtDblSet("Blue Gain", blue_kg);
cxInWdgtEnable("Blue Offset");
blue_ko = 1 - blue_kg;
cxInWdgtDblSet("Blue Offset", blue_ko);
cxInWdgtDisable("Blue Offset");
cxInWdgtDblSet("Blue Offset", blue_ko);
cxInWdgtDblMinMaxSet("Blue Gamma", 0.1, 10.0);
cxInWdgtDblSet("Blue Gamma", blue_gamma);

cxInWdgtDblSet("m1x1", mat1x1);
cxInWdgtDblSet("m1x2", mat1x2);
cxInWdgtDblSet("m1x3", mat1x3);
cxInWdgtDblSet("m2x1", mat2x1);
cxInWdgtDblSet("m2x2", mat2x2);
cxInWdgtDblSet("m2x3", mat2x3);
cxInWdgtDblSet("m3x1", mat3x1);
cxInWdgtDblSet("m3x2", mat3x2);
cxInWdgtDblSet("m3x3", mat3x3);




    /* loop over the data elements */
    for ( i = 0; i < in->dims[1]; i++ )
    {
            for ( j = 0; j < in->dims[0]; j++ )
            {
```

```
temp = red_kg *  ( (double) a[0] / 255.0 +
          red_ko);

if(temp >= 0)
{
rtemp = pow(temp,  red_gamma);
}
else
{
   rtemp = 0.0;
}


temp = green_kg *( (double) a[1] / 255.0 +
          green_ko);

if(temp >= 0)
{
gtemp = pow(temp,  green_gamma);
}
else
{
   gtemp = 0.0;
}



temp = blue_kg *( (double) a[2] / 255.0 +
          blue_ko);

if(temp >= 0)
{
btemp = pow(temp,  blue_gamma);
}
else
{
   btemp = 0.0;
}


b[0] = ((rtemp * mat1x1) +
          (gtemp * mat1x2) +
          (btemp * mat1x3)) / white_X;

b[1] = ((rtemp * mat2x1) +
          (gtemp * mat2x2) +
          (btemp * mat2x3)) / white_Y;

b[2] = ((rtemp * mat3x1) +
          (gtemp * mat3x2) +
          (btemp * mat3x3)) / white_Z;
```

```
                    a += 3;
                    b += 3;


                    }
            }


break;
}

case 1:

if(filename == NULL) return;
if(filename[0] == NULL) return;



if((inputFile = fopen(filename, "r")) == NULL)
{
  cxModAlert("Can't open file!");
  return;
}
else
{
        fprintf(stdout,  "I have made it here past the open.\n");
}



(void) fscanf(inputFile, "%s\n", text);


 (void) fscanf(inputFile, "%lf %lf %lf",&white_X,
        &white_Y, &white_Z);


(void) fscanf(inputFile, "%s\n", text);


 (void) fscanf(inputFile, "%lf  %lf  %lf", &red_ko,
        &red_kg, &red_gamma);
 (void) fscanf(inputFile, "%lf %lf %lf", &green_ko,
        &green_kg, &green_gamma);
 (void) fscanf(inputFile, "%lf %lf %lf", &blue_ko,
        &blue_kg, &blue_gamma);

 (void) fscanf(inputFile, "%s\n", text);

(void) fscanf(inputFile, "%lf %lf %lf",&mat1x1, &mat1x2, &mat1x3);
(void) fscanf(inputFile, "%lf %lf %lf",&mat2x1, &mat2x2, &mat2x3);
(void) fscanf(inputFile, "%lf %lf %lf",&mat3x1, &mat3x2, &mat3x3);
```

```
fprintf(stdout, "I finished the read!\n");

fclose(inputFile);


cxInWdgtDblSet("White Point X", white_X);
 cxInWdgtDblSet("White Point Y", white_Y);
 cxInWdgtDblSet("White Point Z", white_Z);

 cxInWdgtDblMinMaxSet("Red Gain", 0.7, 1.3);
 cxInWdgtDblSet("Red Gain", red_kg);
 cxInWdgtEnable("Red Offset");
 cxInWdgtDblSet("Red Offset", red_ko);
 cxInWdgtDisable("Red Offset");
 cxInWdgtDblMinMaxSet("Red Gamma", 0.1, 10.0);
 cxInWdgtDblSet("Red Gamma", red_gamma);

 cxInWdgtDblMinMaxSet("Green Gain", 0.7, 1.3);
 cxInWdgtDblSet("Green Gain", green_kg);
 cxInWdgtEnable("Green Offset");
 cxInWdgtDblSet("Green Offset", green_ko);
 cxInWdgtDisable("Green Offset");
 cxInWdgtDblMinMaxSet("Green Gamma", 0.1, 10.0);
 cxInWdgtDblSet("Green Gamma", green_gamma);

 cxInWdgtDblMinMaxSet("Blue Gain", 0.7, 1.3);
 cxInWdgtDblSet("Blue Gain", blue_kg);
 cxInWdgtEnable("Blue Offset");
 cxInWdgtDblSet("Blue Offset", blue_ko);
 cxInWdgtDisable("Blue Offset");
 cxInWdgtDblSet("Blue Offset", blue_ko);
 cxInWdgtDblMinMaxSet("Blue Gamma", 0.1, 10.0);
 cxInWdgtDblSet("Blue Gamma", blue_gamma);

 cxInWdgtDblSet("m1x1", mat1x1);
 cxInWdgtDblSet("m1x2", mat1x2);
 cxInWdgtDblSet("m1x3", mat1x3);
 cxInWdgtDblSet("m2x1", mat2x1);
 cxInWdgtDblSet("m2x2", mat2x2);
 cxInWdgtDblSet("m2x3", mat2x3);
 cxInWdgtDblSet("m3x1", mat3x1);
 cxInWdgtDblSet("m3x2", mat3x2);
 cxInWdgtDblSet("m3x3", mat3x3);



        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {


                temp = red_kg * ( (double) a[0] / 255.0 +
```

```
            red_ko);

if(temp >= 0)
{
rtemp = pow(temp,  red_gamma);
}
else
{
   rtemp = 0.0;
}


temp = green_kg *( (double) a[1] / 255.0 +
            green_ko);

if(temp >= 0)
{
gtemp = pow(temp,  green_gamma);
}
else
{
   gtemp = 0.0;
}



temp = blue_kg *( (double) a[2] / 255.0 +
            blue_ko);

if(temp >= 0)
{
btemp = pow(temp,  blue_gamma);
}
else
{
   btemp = 0.0;
}


b[0] = ((rtemp * mat1x1) +
            (gtemp * mat1x2) +
            (btemp * mat1x3)) / white_X;

b[1] = ((rtemp * mat2x1) +
            (gtemp * mat2x2) +
            (btemp * mat2x3))/ white_Y;

b[2] = ((rtemp * mat3x1) +
            (gtemp * mat3x2) +
            (btemp * mat3x3)) / white_Z;




a += 3;
```

```
                b += 3;

                }
        }

  break;

}

}
```

scanner.c *Howtek_Scanner_RGB_to_XYZ_Norm*

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


int ReadLin(char *filename, double Rlin[6],double Glin[6],double Blin[6]);
int ReadMatrix(char *filename,  double CC[11], double MC[11], double YC[11]);
int ReadAbsorp(char *filename,  double kc[36], double km[36], double ky[36],
        double Rg[36]);
int ReadCMFS(char *filename, double xbar[36],double ybar[36], double zbar[36]);
int ReadSource(char *filename,  double s[36]);


void Scanner(cxLattice *in,  cxLattice **out,  double whitePointX,
        double whitePointY,  double whitePointZ, char *filenameAbs,
        char *filenameCMFS,  char *filenameLin,  char *filenameSrc,
        char *filenameMat,  int choice)

{



 cxErrorCode err;
 unsigned char *a;
 double *b;
 double Ref[36], X, Y, Z, Konstant, Cc, Cm, Cy, Rg[36], kc[36], km[36], ky[36],
                xbar[36], ybar[36], zbar[36], s[36], DR, DG, DB, R, G, B,
                Rlin[6], Glin[6], Blin[6], CC[11], MC[11], YC[11];
 double temp;
 int i, j,  k;
```

```
        /* create the new lattice */
        *out = cxLatDataNew(
                2,                  /* number of dimensions */
                in->dims,           /* dimensions array */
                in->data->nDataVar, /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL):


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

        /* Read in the data for the scanner model */

if(choice == 0)
{
  filenameAbs = "/usr/mcsl/explorer/data/Howtek/Scanner.Absorp";
  filenameCMFS = "/usr/mcsl/explorer/data/Howtek/Scanner.CMFS";
  filenameLin = "/usr/mcsl/explorer/data/Howtek/Scanner.Linearization";
  filenameMat = "/usr/mcsl/explorer/data/Howtek/Scanner.Matrix";
  filenameSrc = "/usr/mcsl/explorer/data/Howtek/Scanner.Source";

  cxInWdgtStrSet("Filename(Absorptivities)", filenameAbs);
  cxInWdgtStrSet("Filename(CMFS)", filenameCMFS);
  cxInWdgtStrSet("Filename(Linearization)", filenameLin);
  cxInWdgtStrSet("Filename(Matrix)", filenameMat);
  cxInWdgtStrSet("Filename(Source)", filenameSrc);
}

if(choice == 1)
{

  if(filenameAbs == NULL || filenameCMFS == NULL || filenameLin == NULL ||
    filenameSrc == NULL || filenameMat == NULL)
    {
    return;
    }

  if(filenameAbs[0] == NULL || filenameCMFS[0] == NULL || filenameLin[0] == NULL
    || filenameSrc[0] == NULL || filenameMat[0] == NULL)
    {
    return;
    }
}
        if(ReadMatrix(filenameMat,CC,MC,YC) != 1)
        {
          fprintf(stdout, "Error opening file Scanner.Matrix!\n");
          return;
```

```
}
if(ReadLin(filenameLin, Rlin,Glin,Blin) != 1)
{
   fprintf(stdout, "Error opening file Scanner.Linearization!\n");
   return;
}
if(ReadAbsorp(filenameAbs, kc,km,ky,Rg) != 1)
{
   fprintf(stdout, "Error opening file Scanner.Absorp!\n");
   return;
}
if(ReadCMFS(filenameCMFS, xbar,ybar,zbar) != 1)
{
   fprintf(stdout, "Error opening file Scanner.CMFS!\n");
   return;
}
if(ReadSource(filenameSrc, s) != 1)
{
   fprintf(stdout, "Error opening file Scanner.Source!\n");
   return;
}

/* Calculate tristimulus constant (only once) */

Konstant = 0.0;
for(k=0;k<36;k++){
        Konstant += ybar[k]*s[k];
}
Konstant = 100.0/Konstant;


if(whitePointX == NULL || whitePointY == NULL || whitePointZ == NULL)
{
   return;
}
else
{
   cxInWdgtDblSet("Normalizing White Point X", whitePointX);
   cxInWdgtDblSet("Normalizing White Point Y", whitePointY);
   cxInWdgtDblSet("Normalizing White Point Z", whitePointZ);
}




/* loop over the data elements */
for ( i = 0; i < in->dims[1]; i++ )
{
 for ( j = 0; j < in->dims[0]; j++ )
 {


/* Linearize the scan data */
if(a[0]==0) R=0.0;
else
```

```
        R = -log((double) a[0]/255.0);

        if(a[1]==0) G=0.0;
        else
        G = -log((double) a[1]/255.0);

        if(a[2]==0) B=0.0;
        else
        B = -log((double) a[2]/255.0);
/*
fprintf(stdout, "a[0] = %d a[1] = %d a[2] = %d\n", a[0], a[1], a[2]);
fprintf(stdout, "R = %lf G = %lf B = %lf\n", R,G,B);
*/
        DR = Rlin[0] + Rlin[1]*R + Rlin[2]*R*R + Rlin[3]*R*R*R
                + Rlin[4]*R*R*R*R + Rlin[5]*R*R*R*R*R;

        DG = Glin[0] + Glin[1]*G + Glin[2]*G*G + Glin[3]*G*G*G
                + Glin[4]*G*G*G*G + Glin[5]*G*G*G*G*G;

        DB = Blin[0] + Blin[1]*B + Blin[2]*B*B + Blin[3]*B*B*B
                + Blin[4]*B*B*B*B + Blin[5]*B*B*B*B*B;
/*
fprintf(stdout, "DR = %lf DG = %lf DB = %lf\n", DR,DG,DB);
*/

        /* Calculate the concentrations through the 3x11 !! */
        Cc = CC[0] + CC[1]*DR + CC[2]*DG + CC[3]*DB + CC[4]*DR*DG
                + CC[5]*DR*DB + CC[6]*DG*DB + CC[7]*DR*DG*DB
                + CC[8]*DR*DR + CC[9]*DG*DG + CC[10]*DB*DB;

        Cm = MC[0] + MC[1]*DR + MC[2]*DG + MC[3]*DB + MC[4]*DR*DG
                + MC[5]*DR*DB + MC[6]*DG*DB + MC[7]*DR*DG*DB
                + MC[8]*DR*DR + MC[9]*DG*DG + MC[10]*DB*DB;

        Cy = YC[0] + YC[1]*DR + YC[2]*DG + YC[3]*DB + YC[4]*DR*DG
                + YC[5]*DR*DB + YC[6]*DG*DB + YC[7]*DR*DG*DB
                + YC[8]*DR*DR + YC[9]*DG*DG + YC[10]*DB*DB;

/*
fprintf(stdout, "Cc = %lf Cm = %lf Cy = %lf\n", Cc,Cm,Cy);
*/

        /* Calculate spectral reflectance according to K-M */
        for(k=0;k<36;k++)
    {
                temp = -2.0 * ((Cc*kc[k]) + (Cm*km[k]) + (Cy*ky[k]));

                Ref[k] = Rg[k] * exp(temp);

/*
fprintf(stdout, "Ref[k] = %lf ", Ref[k]);
*/
        }

/*
```

```
fprintf(stdout, "\n");
*/
        /* Calculate Tristimulus values */
        X=0.0;   Y=0.0; Z=0.0;
        for(k=0;k<36;k++)
        {
                X += Ref[k]*s[k]*xbar[k];
                Y += Ref[k]*s[k]*ybar[k];
                Z += Ref[k]*s[k]*zbar[k];
        }



        X = X*Konstant;
        Y = Y*Konstant;
        Z = Z*Konstant;


/*
fprintf(stdout,  "X = %lf Y = %lf Z = %lf K = %lf\n",  X, Y, Z, Konstant);
*/

        /* Put them back into pixels for the image */



        b[0] = X / whitePointX;
        b[1] = Y / whitePointY;
        b[2] = Z / whitePointZ;



                a += 3;
                b += 3;

                }
        }
}

int ReadLin(char *filename, double Rlin[6],double Glin[6],double Blin[6])
{
        int i;
        FILE *datafile;

        if((datafile = fopen(filename, "r")) == NULL)
        {
          fprintf(stdout,  "Fopen failed for Linearization!\n");
          return(0);
        }

        for(i=0;i<6;i++){
                fscanf(datafile,"%lf",&Rlin[i]);
        }
        for(i=0;i<6;i++){
                fscanf(datafile,"%lf",&Glin[i]);
```

```
        }
        for(i=0;i<6;i++){
                fscanf(datafile,"%lf",&Blin[i]);
        }
        fclose(datafile);
        return(1);
}


int ReadMatrix(char *filename,  double CC[11], double MC[11], double YC[11])
{
        int i;
        FILE *datafile;


        if((datafile = fopen(filename, "r")) == NULL)
        {
          fprintf(stdout,  "Fopen failed for Matrix!\n");
          return(0);
        }

        for(i=0;i<11;i++){
                fscanf(datafile,"%lf",&CC[i]);
        }
        for(i=0;i<11;i++){
                fscanf(datafile,"%lf",&MC[i]);
        }
        for(i=0;i<11;i++){
                fscanf(datafile,"%lf",&YC[i]);
        }
        fclose(datafile);
        return(1);
}


int ReadAbsorp(char *filename,  double kc[36], double km[36], double ky[36],
        double Rg[36])
{
        int i;
        FILE *datafile;

        if((datafile = fopen(filename, "r")) == NULL)
        {
          fprintf(stdout,  "Fopen failed for Absorp!\n");
          return(0);
        }

        for(i=0;i<36;i++){
                fscanf(datafile,"%lf%lf%lf%lf",&kc[i],&km[i],&ky[i],&Rg[i]);
        }
        return(1);
}


int ReadCMFS(char *filename, double xbar[36],double ybar[36], double zbar[36])
```

```
{
        int i;
        FILE *datafile;

        if((datafile = fopen(filename, "r")) == NULL)
        {
          fprintf(stdout, "Fopen failed for CMFS!\n");
          return(0);
        }

        for(i=0;i<36;i++){
                fscanf(datafile,"%lf%lf%lf",&xbar[i],&ybar[i],&zbar[i]);
        }
        return(1);
}


int ReadSource(char *filename, double s[36])
{
        int i;
        FILE *datafile;

        if((datafile = fopen("/usr/people/hauf/Explorer/Thesis/Scanner/Howtek/Scanner.Source", "r")) ==
NULL)
        {
          fprintf(stdout, "Fopen failed for Source!\n");
          return(0);
        }

        for(i=0;i<36;i++){
                fscanf(datafile,"%lf",&s[i]);
        }
        return(1);
}
```

solitaire_xyz_rgb.c *XYZ_Norm_to_Solitaire_RGB*

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>

extern int SolLoad(char *filenameLuts, char *filenameWghts, char *filenameVect,
            double R_C[256], double G_C[256], double B_C[256], double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31]);

extern int SolXYZ_RGB2(double XYZ[3],int RGB[3], double R_C[256], double G_C[256],
        double B_C[256], double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
```

```
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31]);


void SolitaireXYZ_RGB(cxLattice *in, cxLattice **out, double whitePointX,
            double whitePointY, double whitePointZ, char *filenameLuts,
            char *filenameWghts, char *filenameVect, int choice)

{



 cxErrorCode err;
 double *a;
 unsigned char *b;
 int RGB[3];
 double XYZ[3];
 double R_C[256], G_C[256], B_C[256];
 double R_M[256], G_M[256], B_M[256];
 double R_Y[256], G_Y[256], B_Y[256];
 double xwt[31], ywt[31], zwt[31], cvect[31], mvect[31], yvect[31], dmin[31];
 int i, j, k;




        /* create the new lattice */
        *out = cxLatDataNew(
                2,                  /* number of dimensions */
                in->dims,                   /* dimensions array */
                in->data->nDataVar,         /* number of data variables */
                cx_prim_byte);   /* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);



        if(whitePointX == NULL || whitePointY == NULL || whitePointZ == NULL)
        {
          return;
        }
        else
        {
          cxInWdgtDblSet("Normalizing White Point X", whitePointX);
          cxInWdgtDblSet("Normalizing White Point Y", whitePointY);
          cxInWdgtDblSet("Normalizing White Point Z", whitePointZ);
        }

if(choice == 0)
```

```
{
  filenameLuts = "/usr/mcsl/explorer/data/Solitaire/iiluts.txt";
  filenameVect = "/usr/mcsl/explorer/data/Solitaire/vectors.txt";
  filenameWghts = "/usr/mcsl/explorer/data/Solitaire/tsv.wts";

  cxInWdgtStrSet("Filename(Luts)", filenameLuts);
  cxInWdgtStrSet("Filename(Weights)", filenameWghts);
  cxInWdgtStrSet("Filename(Vectors)", filenameVect);

  SolLoad(filenameLuts, filenameWghts, filenameVect,
          R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
          cvect, mvect, yvect, dmin, xwt, ywt, zwt);

}

if(choice == 1)
{

  if(filenameLuts == NULL || filenameWghts == NULL || filenameVect == NULL)
    {
    return;
    }

  if(filenameLuts[0] == NULL || filenameWghts[0] == NULL ||
       filenameVect[0] == NULL)
    {
    return;
    }

  if((SolLoad(filenameLuts, filenameWghts, filenameVect,
          R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
          cvect, mvect, yvect, dmin, xwt, ywt, zwt)) != 1)
  {
       fprintf(stdout, "Error opening files\n");
       return;
  }

}

       /* loop over the data elements */
       for ( i = 0; i < in->dims[1]; i++ )
       {
        for ( j = 0; j < in->dims[0]; j++ )
        {

         XYZ[0] = a[0] * whitePointX;
         XYZ[1] = a[1] * whitePointY;
         XYZ[2] = a[2] * whitePointZ;


         SolXYZ_RGB2(XYZ,RGB,R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
              cvect, mvect, yvect, dmin, xwt, ywt, zwt);
```

```
            b[0] = (unsigned char) RGB[0];
            b[1] = (unsigned char) RGB[1];
            b[2] = (unsigned char) RGB[2];

            a += 3;
        b += 3;




            }
          fprintf(stdout,  "i = %d\n",  i);
          }
}
```

SolXYZ_RGB2.c *XYZ_Norm_to_Solitaire_RGB*

```
/*      Mark D. Fairchild
        Program to Calculate Solitaire RGB from XYZ
        Using the "Brute Force" iteration technique
        DATE: 6/16/93
*/
/*
        Christopher R. Hauf
        Program modified for use in Explorer framework including change to
        doubles and ANSI C function syntax
        DATE: 2/5/96
        Program modified to move loading of functions to main calling function
        DATE: 2/11/97
 */


#include<stdio.h>
#include<math.h>



int SolRGB_XYZ(int RGB[3], double XYZ[3], double R_C[256], double G_C[256], double B_C[256],
double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31]);

#define MAXITER 300

int SolXYZ_RGB2(double XYZ[3],int RGB[3], double R_C[256], double G_C[256],
        double B_C[256], double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31])
{
```

```
double XYZrp[3], XYZrm[3], XYZgp[3], XYZgm[3], XYZbp[3], XYZbm[3];
double dXrp, dXrm;
double dYgp, dYgm;
double dZbp, dZbm;
int flag, i, STEP, j;




RGB[0] = RGB[1] = RGB[2] = 128;

STEP = 1;

i = 0;
while(i<MAXITER){
        RGB[0] = RGB[0] + STEP;
        SolRGB_XYZ(RGB, XYZrp, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);

        RGB[0] = RGB[0] - 2*STEP;
        SolRGB_XYZ(RGB, XYZrm, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);
        RGB[0] = RGB[0] + STEP;
        RGB[1] = RGB[1] + STEP;
        SolRGB_XYZ(RGB, XYZgp, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);
        RGB[1] = RGB[1] - 2*STEP;
        SolRGB_XYZ(RGB, XYZgm, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);
        RGB[1] = RGB[1] + STEP;
        RGB[2] = RGB[2] + STEP;
        SolRGB_XYZ(RGB, XYZbp, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);
        RGB[2] = RGB[2] - 2*STEP;
        SolRGB_XYZ(RGB, XYZbm, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);
        RGB[2] = RGB[2] + STEP;

        dXrp = XYZrp[0] - XYZ[0];
        dXrm = XYZrm[0] - XYZ[0];

        dYgp = XYZgp[1] - XYZ[1];
        dYgm = XYZgm[1] - XYZ[1];

        dZbp = XYZbp[2] - XYZ[2];
        dZbm = XYZbm[2] - XYZ[2];

        flag = 0;

        if (dXrp*dXrm<0) flag = flag+1;
        if (dYgp*dYgm<0) flag = flag+1;
        if (dZbp*dZbm<0) flag = flag+1;
        if (flag == 3 && STEP == 5 ) {
                STEP =1;
```

```
                i=0;
                }
        else if (flag == 3 && STEP == 1) return(1);

        dXrp =  fabs(dXrp);
        dXrm =  fabs(dXrm);

        dYgp =  fabs(dYgp);
        dYgm =  fabs(dYgm);

        dZbp =  fabs(dZbp);
        dZbm =  fabs(dZbm);

        if (dXrp>dXrm) RGB[0] = RGB[0]-STEP;
        if (dXrp<dXrm) RGB[0] = RGB[0]+STEP;

        if (dYgp<dYgm) RGB[1] = RGB[1]+STEP;
        if (dYgp>dYgm) RGB[1] = RGB[1]-STEP;

        if (dZbp>dZbm) RGB[2] = RGB[2]-STEP;
        if (dZbp<dZbm ) RGB[2] = RGB[2]+STEP;



        if (i<MAXITER){
                if(RGB[0]>255-STEP) RGB[0]=255-STEP;
                if(RGB[0]<0+STEP) RGB[0]=0+STEP;
                if(RGB[1]>255-STEP) RGB[1]=255-STEP;
                if(RGB[1]<0+STEP) RGB[1]=0+STEP;
                if(RGB[2]>255-STEP) RGB[2]=255-STEP;
                if(RGB[2]<0+STEP) RGB[2]=0+STEP;
                }
        i++;

        }


fprintf(stdout, "MAXITER exceeded\n");

return(1);
}
```

SolRGB_XYZ.c *XYZ_Norm_to_Solitaire_RGB*
Solitaire_RGB_to_XYZ_Norm

```
/*      Mark D. Fairchild
        Program to Calculate Solitaire RGB from XYZ
        Using the "Brute Force" iteration technique
        DATE: 6/16/93
*/
/*
        Christopher R. Hauf
        Program modified for use in Explorer framework including change to
        doubles and ANSI C function syntax
```

```
        DATE: 2/5/96
        Program modified to move loading of functions to main calling function
        DATE: 2/11/97
*/


#include<stdio.h>
#include<math.h>



int SolRGB_XYZ(int RGB[3], double XYZ[3], double R_C[256], double G_C[256], double B_C[256],
double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31]);

#define MAXITER 300

int SolXYZ_RGB2(double XYZ[3],int RGB[3], double R_C[256], double G_C[256],
        double B_C[256], double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31])
{

double XYZrp[3], XYZrm[3], XYZgp[3], XYZgm[3], XYZbp[3], XYZbm[3];
double dXrp, dXrm;
double dYgp, dYgm;
double dZbp, dZbm;
int flag, i, STEP, j;




RGB[0] = RGB[1] = RGB[2] = 128;

STEP = 1;

i = 0;
while(i<MAXITER){
        RGB[0] = RGB[0] + STEP;
        SolRGB_XYZ(RGB, XYZrp, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);

        RGB[0] = RGB[0] - 2*STEP;
        SolRGB_XYZ(RGB, XYZrm, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);
        RGB[0] = RGB[0] + STEP;
        RGB[1] = RGB[1] + STEP;
        SolRGB_XYZ(RGB, XYZgp, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
         cvect, mvect, yvect, dmin, xwt, ywt, zwt);
```

```
RGB[1] = RGB[1] - 2*STEP;
SolRGB_XYZ(RGB, XYZgm, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
 cvect, mvect, yvect, dmin, xwt, ywt, zwt);
RGB[1] = RGB[1] + STEP;
RGB[2] = RGB[2] + STEP;
SolRGB_XYZ(RGB, XYZbp, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
 cvect, mvect, yvect, dmin, xwt, ywt, zwt);
RGB[2] = RGB[2] - 2*STEP;
SolRGB_XYZ(RGB, XYZbm, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
 cvect, mvect, yvect, dmin, xwt, ywt, zwt);
RGB[2] = RGB[2] + STEP;

dXrp = XYZrp[0] - XYZ[0];
dXrm = XYZrm[0] - XYZ[0];

dYgp = XYZgp[1] - XYZ[1];
dYgm = XYZgm[1] - XYZ[1];

dZbp = XYZbp[2] - XYZ[2];
dZbm = XYZbm[2] - XYZ[2];

flag = 0;

if (dXrp*dXrm<0) flag = flag+1;
if (dYgp*dYgm<0) flag = flag+1;
if (dZbp*dZbm<0) flag = flag+1;
if (flag == 3 && STEP == 5 ) {
        STEP =1;
        i=0;
        }
else if (flag == 3 && STEP == 1) return(1);

dXrp =  fabs(dXrp);
dXrm =  fabs(dXrm);

dYgp =  fabs(dYgp);
dYgm =  fabs(dYgm);

dZbp =  fabs(dZbp);
dZbm =  fabs(dZbm);

if (dXrp>dXrm) RGB[0] = RGB[0]-STEP;
if (dXrp<dXrm) RGB[0] = RGB[0]+STEP;

if (dYgp<dYgm) RGB[1] = RGB[1]+STEP;
if (dYgp>dYgm) RGB[1] = RGB[1]-STEP;

if (dZbp>dZbm) RGB[2] = RGB[2]-STEP;
if (dZbp<dZbm ) RGB[2] = RGB[2]+STEP;



if (i<MAXITER){
        if(RGB[0]>255-STEP) RGB[0]=255-STEP;
        if(RGB[0]<0+STEP) RGB[0]=0+STEP;
```

```
                if(RGB[1]>255-STEP) RGB[1]=255-STEP;
                if(RGB[1]<0+STEP) RGB[1]=0+STEP;
                if(RGB[2]>255-STEP) RGB[2]=255-STEP;
                if(RGB[2]<0+STEP) RGB[2]=0+STEP;
                }
        i++;

        }


fprintf(stdout, "MAXITER exceeded\n");

return(1);
}
```

SolLoad.c *XYZ_Norm_to_Solitaire_RGB*
Solitaire_RGB_to_XYZ_Norm

```
#include<stdio.h>

int SolLoad(char *filenameLuts, char *filenameWghts, char*filenameVect,
            double R_C[256], double G_C[256], double B_C[256], double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31])
{
double junk;
int i, dc;
FILE *lutsfile, *vectorsfile, *wtsfile;

if((lutsfile = fopen(filenameLuts, "r")) == NULL)
{
   fprintf(stdout,  "Could not open Luts file!\n");
   return(0);
}
if((vectorsfile = fopen(filenameVect , "r")) == NULL)
{
   fprintf(stdout,  "Could not open Vectors file!\n");
   return(0);
}
if((wtsfile = fopen(filenameWghts, "r")) == NULL)
{
   fprintf(stdout,  "Could not open Weights file!\n");
   return(0);
}

i = 0;
while(i<31){
        fscanf(vectorsfile, "%lf%lf%lf%lf%lf", &junk, &cvect[i], &mvect[i],
                &yvect[i], &dmin[i]);
        fscanf(wtsfile, "%lf%lf%lf", &xwt[i], &ywt[i], &zwt[i]);

        i++;
        }
```

```
i = 0;
while(i<256){
        fscanf(lutsfile, "%d%lf%lf%lf%lf%lf%lf%lf%lf", &dc,
                                                &R_C[i], &G_C[i], &B_C[i],
                                                &R_M[i], &G_M[i], &B_M[i],
                                                &R_Y[i], &G_Y[i], &B_Y[i]);
    fprintf(stdout, "rclut %lf\n", R_C[i]);
        i++;
        }

fclose(lutsfile);
fclose(vectorsfile);
fclose(wtsfile);

return(1);
}
```

solitaire_rgb_xyz.c *Solitaire_RGB_to_XYZ_Norm*

```
#include<stdio.h>
#include<math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


int SolLoad(char *filenameLuts,  char *filenameWghts, char *filenameVect,
            double R_C[256], double G_C[256], double B_C[256], double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31]);

int SolRGB_XYZ(int RGB[3], double XYZ[3], double R_C[256], double G_C[256], double B_C[256],
double R_M[256],
            double G_M[256], double B_M[256], double R_Y[256], double G_Y[256],
            double B_Y[256], double cvect[31], double mvect[31],
            double yvect[31], double dmin[31], double xwt[31],
            double ywt[31], double zwt[31]);

void SolitaireRGB_XYZ(cxLattice *in,  cxLattice **out,  double whitePointX,
            double whitePointY,  double whitePointZ, char *filenameLuts,
            char *filenameWghts,  char *filenameVect,  int choice)

{



 cxErrorCode err;
 unsigned char *a;
 double *b;
 int RGB[3];
 double XYZ[3];
```

```
double R_C[256], G_C[256], B_C[256];
double R_M[256], G_M[256], B_M[256];
double R_Y[256], G_Y[256], B_Y[256];
double xwt[31], ywt[31], zwt[31], cvect[31], mvect[31], yvect[31], dmin[31];

int i, j,  k;




        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,                       /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

        /* Read in the data for the solitaire model */

if(choice == 0)
{
   filenameLuts = "/usr/mcsl/explorer/data/Solitaire/iiluts.txt";
   filenameVect = "/usr/mcsl/explorer/data/Solitaire/vectors.txt";
   filenameWghts = "/usr/mcsl/explorer/data/Solitaire/tsv.wts";

   cxInWdgtStrSet("Filename(Luts)", filenameLuts);
   cxInWdgtStrSet("Filename(Weights)", filenameWghts);
   cxInWdgtStrSet("Filename(Vectors)", filenameVect);

   SolLoad(filenameLuts, filenameWghts, filenameVect,
            R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
            cvect, mvect, yvect, dmin, xwt, ywt, zwt);

}

if(choice == 1)
{

   if(filenameLuts == NULL || filenameWghts == NULL || filenameVect == NULL)
     {
     return;
     }

   if(filenameLuts[0] == NULL || filenameWghts[0] == NULL ||
         filenameVect[0] == NULL)
     {
     return;
```

```
    }

  if((SolLoad(filenameLuts, filenameWghts, filenameVect,
          R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
          cvect, mvect, yvect, dmin, xwt, ywt, zwt)) != 1)
  {
        fprintf(stdout, "Error opening files\n");
        return;
  }


}

        if(whitePointX == NULL || whitePointY == NULL || whitePointZ == NULL)
        {
          return;
        }
        else
        {
          cxInWdgtDblSet("Normalizing White Point X", whitePointX);
          cxInWdgtDblSet("Normalizing White Point Y", whitePointY);
          cxInWdgtDblSet("Normalizing White Point Z", whitePointZ);
        }




        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
         for ( j = 0; j < in->dims[0]; j++ )
         {

          RGB[0] = (int) a[0];
          RGB[1] = (int) a[1];
          RGB[2] = (int) a[2];


          SolRGB_XYZ(RGB, XYZ, R_C, G_C, B_C, R_M, G_M, B_M, R_Y, G_Y, B_Y,
        cvect, mvect, yvect, dmin, xwt, ywt, zwt);




          b[0] = XYZ[0] / whitePointX;
          b[1] = XYZ[1] / whitePointY;
          b[2] = XYZ[2] / whitePointZ;

      a += 3;
          b += 3;


         }
        }
}
```

8.2 Color appearance models

XYZ2H93.c *XYZ_Norm_to_Hunt93*

```
/*

   XYZ2H93.c

   This function takes in normalized XYZ tristimulus values, unnormalizes
   them and proceeds to calculate JCh for Hunt's 93 color appearance model.

   b[0] = J
   b[1] = C
   b[2] = h

   Christopher R. Hauf
   December 29, 1995

   Based on H93PreCalc.c by Mike Stokes &Mark Fairchild.  See below

   Revisions:

   5/13/96 - CRH - Added precalc flag

*/

/*************************************************************************/
/*                                                                       */
/* file: H93PreCalc.c */
/* description: Precalculate passed variables for XYZ_H93 & H93_XYZ */
/*                                                                       */
/* author:  Mike Stokes &  Mark Fairchild                                */
/*       Date: 1/27/93 */
/*           Mike Stokes (mdspci@judd.cis.rit.edu)           */
/*           Munsell Color Science Laboratory (MCSL)          */
/*           Color Science Department                      */
/*           Center for Imaging Science                    */
/*           Rochester Institute of Technology               */
/*           Rochester, NY  14623                        */
/*                                                                       */
/*************************************************************************/



#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>
```

```
#define PI 3.14159265
#define SOFTCOPY 0
#define HARDCOPY 1



void XYZ_Norm_to_H93(cxLattice *in, cxLattice **out, double Xnr, double Ynr,
        double Znr, double Xn, double Yn, double Zn, double LW,
                double T,  double Yb, int choice,  int softhard, int prec alc,
                cxLattice **model_data)
{


  double *a;
  double *b;
  double *c;
  cxErrorCode err;
  int i,  j;
  double temp, LA, LAS, Nc, Nb, Ncb, Nbb, rhoW, gammaW, betaW;
  double FL, Frho, Fgamma, Fbeta, rhoD, gammaD, betaD, Brho, Bgamma, Bbeta;
  double rhoA, gammaA, betaA, AA, C1, C2, C3, t, tprime, es;
  double rhoAW, gammaAW, betaAW, AAW, C1W, C2W, C3W, tW, tprimeW, hsW, esW;
  double MYB, MRG, M, FLS, BS, AS, A;
  double Ft, MYBW, MRGW, MW, BSW, ASW, AW, N1, N2, QW, z, YBYW;
  double rho, gamma, beta, s, Q, hs;
  double tmpBS, tmp1, tmp2, tmp3;
  double XYZ[3],  JCh[3];
  long dimensions[1] = 24;


        /* create the new image lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,                       /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /*Create new 1D lattice containing Hunt model parameters for use
          in the inverse model*/

        *model_data = cxLatDataNew(
                1,                      /* number of dimensions */
                dimensions,                     /* dimensions array */
                1,      /* number of data variables */
                cx_prim_double);/* data type */


        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);
    /* set up new coordinates for model parameters lattice */
        cxLatPtrSet(*model_data,NULL,NULL, cxCoordDefaultNew(1, dimensions), NULL);


        /* extract the data pointers */
```

```
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
        cxLatPtrGet(*model_data,NULL, (void **)&c, NULL, NULL);

        if(Xnr == NULL || Ynr == NULL || Znr == NULL || Xn == NULL || Yn == NULL
      || Zn == NULL || LW == NULL || T == NULL || Yb == NULL)
        {
          return;
        }
        else
        {
          cxInWdgtDblSet("N White Point X", Xnr);
          cxInWdgtDblSet("N White Point Y", Ynr);
          cxInWdgtDblSet("N White Point Z", Znr);
          cxInWdgtDblSet("White Point X", Xn);
          cxInWdgtDblSet("White Point Y", Yn);
          cxInWdgtDblSet("White Point Z", Zn);
          cxInWdgtDblSet("White Luminance", LW);
          cxInWdgtDblSet("Approx CCT of Source", T);
          cxInWdgtDblSet("Bk Rel Per Luminance", Yb);
        }



/* Calculate LA */
  LA = LW / 5.0;

/* Calculate LAS */
  LAS = 2.26*LA*pow((T/4000.0 - 0.4),(1.0/3.0));

/* Specify Nc and Nb */
  switch (choice)
        {
        case 0:
        Nc = 1.0;
        Nb = 300.0;
        break;
        case 1:
        Nc = 1.0;
        Nb = 75.0;
        break;
        case 2:
        Nc = 1.0;
        Nb = 25.0;
        break;
        case 3:
        Nc = 0.7;
        Nb = 10.0;
        break;
        case 4:
        Nc = 0.75;
        Nb = 5;
        break;
        case 5:
```

```
        Nc = 0.7;
        Nb = 75; /* we're notsure what Nb should be in this case */
        break;
        }

/* Calculate Ncb and Nbb */
  Ncb = 0.725*pow((Yn/Yb),0.2);
  Nbb = 0.725*pow((Yn/Yb),0.2);

/* Calculate rhoW, gammaW. and beteW */
  rhoW = 0.38971*Xn + 0.68898*Yn - 0.07868*Zn;
  gammaW = -0.22981*Xn + 1.18340*Yn + 0.04641*Zn;
  betaW = 1.0*Zn;

/* Caclulate FL */
  temp = 1.0/(5.0*LA + 1.0);
  FL = 0.2*pow(temp,4.0)*5.0*LA + 0.1*pow((1.0-pow(temp,4.0)),2.0)*pow(5.0*LA,(1.0/3.0));

/* Calculate Frho, Fgamma, Fbeta */
  if(softhard == HARDCOPY) {
        Frho = 1.0;
        Fgamma = 1.0;
        Fbeta = 1.0;
        }
  else {
        temp = (3.0*rhoW)/(rhoW + gammaW + betaW);
        Frho = (1.0+pow(LA,(1.0/3.0))+temp)/(1.0+pow(LA,(1.0/3.0))+(1.0/temp));
        temp = (3.0*gammaW)/(rhoW + gammaW + betaW);
        Fgamma = (1.0+pow(LA,(1.0/3.0))+temp)/(1.0+pow(LA,(1.0/3.0))+(1.0/temp));
        temp = (3.0*betaW)/(rhoW + gammaW + betaW);
        Fbeta = (1.0+pow(LA,(1.0/3.0))+temp)/(1.0+pow(LA,(1.0/3.0))+(1.0/temp));
        }

/* Calculate rhoD, gammaD, betaD */
  if(softhard == HARDCOPY) {
        rhoD = 0.0;
        gammaD = 0.0;
        betaD = 0.0;
        }
  else {
        rhoD = 40.0*(pow(((Yb/Yn)*FL*Fgamma),0.73)/(pow(((Yb/Yn)*FL*Fgamma),0.73)+2.0)) -
40.0*(pow(((Yb/Yn)*FL*Frho),0.73)/(pow(((Yb/Yn)*FL*Frho),0.73)+2.0));
        gammaD = 0.0;
        betaD = 40.0*(pow(((Yb/Yn)*FL*Fgamma),0.73)/(pow(((Yb/Yn)*FL*Fgamma),0.73)+2.0))
40.0*(pow(((Yb/Yn)*FL*Fbeta),0.73)/(pow(((Yb/Yn)*FL*Fbeta),0.73)+2.0));
        }

/* Calculate Brho, Bgamma, Bbeta */
  Brho = pow(10.0,7.0)/(pow(10.0,7.0)+5.0*LA*(rhoW/100.0));
  Bgamma = pow(10.0,7.0)/(pow(10.0,7.0)+5.0*LA*(gammaW/100.0));
  Bbeta = pow(10.0,7.0)/(pow(10.0,7.0)+5.0*LA*(betaW/100.0));

/* Calculate rhoAW, gammaAW, betaAW */
  rhoAW = Brho*(40.0*(pow((FL*Frho),0.73)/(pow((FL*Frho),0.73)+2.0))+rhoD) + 1.0;
```

```
  gammaAW = Bgamma*(40.0*(pow((FL*Fgamma),0.73)/(pow((FL*Fgamma),0.73)+2.0))+gammaD) +
1.0;
  betaAW = Bbeta*(40.0*(pow((FL*Fbeta),0.73)/(pow((FL*Fbeta),0.73)+2.0))+betaD) + 1.0;

/* Calculate AAW */
  AAW = 2.0*rhoAW + gammaAW + 0.05*betaAW - 3.05 + 1.0;

/* Calculate C1W, C2W, C3W */
  C1W = rhoAW - gammaAW;
  C2W = gammaAW - betaAW;
  C3W = betaAW - rhoAW;

/* Calculate tW and tprimeW */
  tW = 0.5*(C2W - C3W)/4.5;
  tprimeW = C1W - (C2W/11.0);

/* Calculate hsW */
  hsW = atan2(tW,tprimeW)*(180/PI);
  if(hsW<0) hsW = 360 + hsW;

/* calculate esWW */
  if(hsW >= 0.0 && hsW < 20.14)
          esW = 1.2 - 0.4*(hsW+122.47)/142.61;

  else if(hsW >= 20.14 && hsW < 90.0)
   esW = 0.8 - 0.1*(hsW-20.14)/69.86;

  else if(hsW >= 90.0 && hsW < 164.25)
           esW = 0.7 + 0.3*(hsW-90.0)/74.25;

  else if(hsW >= 164.25 && hsW < 237.53)
           esW = 1.0 + 0.2*(hsW-164.25)/73.28;

  else
   esW = 1.2 - 0.4*(hsW-237.53)/142.61;

/* Calculate Ft */
  Ft = LA / (LA + 0.1);

/* Calcualte MYBW, MRGW, and MW */
  MYBW = 100.0*tW*esW*(10.0/13.0)*Nc*Ncb*Ft;
  MRGW = 100.0*tprimeW*esW*(10.0/13.0)*Nc*Ncb;
  MW = pow((MYBW*MYBW + MRGW*MRGW),0.5);

/* Calculate FLS */
  temp = 0.00001/(5.0*LAS/2.26 + 0.00001);
  FLS = (3800.0*temp*temp*5.0*LAS/2.26) +
                                    (0.2*pow((1.0 - temp*temp), 4.0))*pow((5.0*LAS/2.26), 1.0/6.0);;

/* Calculate BSW */
  BSW = 0.5/(1.0+0.3*(pow(((5.0*LAS/2.26)*1.0),0.3)))
                          + 0.5/(1.0+5.0*(5.0*LAS/2.26));

/* Calculate ASW */
  ASW = BSW*3.05*(40.0*(pow(FLS,0.73)/(pow(FLS,0.73)+2.0))) + 0.3;
```

```
/* Calculate AW */
  AW = Nbb*(AAW - 1.0 + ASW - 0.3 + 1.0440365);

/* Calculate N1 and N2 */
  N1 = pow(7.0*AW,0.5)/(5.33*pow(Nb,0.13));
  N2 = 7.0*AW*pow(Nb,0.362)/200.0;

/* Calculate QW */
  QW = pow((7.0*(AW + MW/100.0)),0.6)*N1 - N2;

/* Calculate z and YBYW */
  z = 1.0 + pow((Yb/Yn),0.5);
  YBYW = Yb/Yn;

/* Load up the model_data lattice*/

  *c =  Brho;
  *(c+1) =  Bgamma;
  *(c+2) =  Bbeta;
  *(c+3) =  FL;
  *(c+4) =  Frho;
  *(c+5) =  Fgamma;
  *(c+6) =  Fbeta;
  *(c+7) =  rhoW;
  *(c+8) =  gammaW;
  *(c+9) =  betaW;
  *(c+10) = rhoD;
  *(c+11) = gammaD;
  *(c+12) = betaD;
  *(c+13) = Nc;
  *(c+14) = Ncb;
  *(c+15) = Ft;
  *(c+16) = LAS;
  *(c+17) = FLS;
  *(c+18) = Nbb;
  *(c+19) = N1;
  *(c+20) = N2;
  *(c+21) = QW;
  *(c+22) = z;
  *(c+23) = YBYW;


if(precalc == 0)
{
   return;

}
else
{

/* loop over the data elements */
 for ( i = 0; i < in->dims[1]; i++ )
 {
  for ( j = 0; j < in->dims[0]; j++ )
```

```
{

  XYZ[0] = a[0] * Xnr;
  XYZ[1] = a[1] * Ynr;
  XYZ[2] = a[2] * Znr;


/* Calculate rho, gamma and beta */

  rho = 0.38971*XYZ[0] + 0.68898*XYZ[1] - 0.07868*XYZ[2];
  gamma = -0.22981*XYZ[0] + 1.18340*XYZ[1] + 0.04641*XYZ[2];
  beta = 1.0*XYZ[2];

/* Calculate rhoA, gammaA, betaA */
  if (FL*Frho*rho/rhoW <= 0) {
    tmp1 = 0.0;
  }
  else {
    tmp1 = pow((FL*Frho*rho/rhoW), 0.73);

  }

  if (FL*Fgamma*gamma/gammaW <= 0) {
    tmp2 = 0.0;
  }
  else {
    tmp2 = pow((FL*Fgamma*gamma/gammaW),0.73);

  }


  if (FL*Fbeta*beta/betaW <= 0) {
    tmp3 = 0.0;
  }
  else {
    tmp3 = pow((FL*Fbeta*beta/betaW),0.73);

  }

  rhoA = Brho*(40.0 * (tmp1 / (tmp1+2.0) ) + rhoD) + 1.0;
  gammaA = Bgamma*(40.0*(tmp2 / (tmp2+2.0) ) + gammaD) + 1.0;
  betaA = Bbeta*(40.0*(tmp3 / (tmp3 + 2.0) ) + betaD) + 1.0;

/* Calculate AA */
  AA = 2.0*rhoA + gammaA + 0.05*betaA - 3.05 + 1.0;

/* Calculate C1, C2, C3 */
  C1 = rhoA - gammaA;
  C2 = gammaA - betaA;
  C3 = betaA - rhoA;

/* Calculate t and tprime */
  t = 0.5*(C2 - C3)/4.5;
  tprime = C1 - (C2/11.0);
```

```
/* Calculate hs */
  errno = 0;
  hs = atan2(t,tprime)*(180/PI);
  if (errno) {
    fprintf(stdout,  "XYZ %f %f %f\n", XYZ[0], XYZ[1], XYZ[2]);
    fprintf(stdout, "errno = %d, t = %f, tprime = %f\n", errno, t, tprime);
    fprintf(stdout, "fXYZ_Hunt93 : atan2 error at pixel %d, %d\n",  i,  j);
   }
  if(hs<0) hs = 360 + hs;
  JCh[2] = hs;

/* calculate es */
  if(hs >= 0.0 && hs < 20.14)
     es = 1.2 - 0.4*(hs+122.47)/142.61;

  else if(hs >= 20.14 && hs < 90.0)
    es = 0.8 - 0.1*(hs-20.14)/69.86;

  else if(hs >= 90.0 && hs < 164.25)
    es = 0.7 + 0.3*(hs-90.0)/74.25;

  else if(hs >= 164.25 && hs < 237.53)
    es = 1.0 + 0.2*(hs-164.25)/73.28;

  else
    es = 1.2 - 0.4*(hs-237.53)/142.61;

/* Calcualte MYB, MRG, and M */
  MYB = 100.0*t*es*(10.0/13.0)*Nc*Ncb*Ft;
  MRG = 100.0*tprime*es*(10.0/13.0)*Nc*Ncb;
  M = pow((MYB*MYB + MRG*MRG),0.5);


/* Calculate s */
  s = (50.0 * M) / (rhoA + gammaA + betaA);

/* Calculate BS */
  tmpBS = pow(((5.0*LAS/2.26) * (XYZ[1]/100.0)), 0.3);

  BS = 0.5 / (1.0+0.3*tmpBS) + 0.5/(1.0+5.0*(5.0*LAS/2.26));

/* Calculate AS */
  tmp1 = pow((FLS*((double)XYZ[1]/100.0)),0.73);


  AS = BS*3.05 * (40.0*(tmp1 / (tmp1 + 2.0))) + 0.3;

/* Calculate A */
  A = Nbb*(AA - 1.0 + AS - 0.3 + 1.0440365);

/* Calculate Q */
  tmpBS = 7.0*(A + M/100.0);
  tmp1 = pow(tmpBS, 0.6);
```

```
    Q = tmp1 * N1 - N2;

  /* Calculate J */
    if (Q >= 0.0) {
     tmp2 = pow((Q/QW), z);

     JCh[0] = 100.0 * tmp2;
    }
    else {
     JCh[0] = 0.0;
    }

  /* Calculate Cb */
    tmp1 = pow(s, 0.69);


    if (Q >= 0.0) {
     tmp2 = pow((Q/QW),YBYW);

    }
    else {
     tmp2 = 0.0;
    }

   tmp3 = pow(0.29, YBYW);


   JCh[1] =  2.44*tmp1*tmp2*(1.64 - tmp3);


   b[0] = JCh[0];
   b[1] = JCh[1];
   b[2] = JCh[2];



                    a += 3;
                    b += 3;

                    }

          }

}
}
```

H93_to_XYZ.c ***Hunt92_to_XYZ_Norm***

```
/*

  H93_to_XYZ.c
```

```
   This file is the main calling function for Hunt93 JCh to XYZ.  This
   function builds on modified CRS code originally written by Mike
   Stokes and Mark Fairchild.


   Christopher Hauf
   December 28, 1995

*/

#include "crs.h"
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <stdlib.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>



void Hunt93_to_XYZ_Norm(cxLattice *in,  cxLattice **out, cxLattice *model_data,
                        double Xn,  double Yn,  double Zn)
{



 cxErrorCode err;
 double *a;
 double *b;
 double *c;
 float XYZ[3], JCh[3];
 int i, j, return_val,  values;
 xfcn_str adapt_model;
 xfcn_ptr adapt_model_ptr;
 double *tmp;




        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,                       /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);
```

```
        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(model_data,NULL,(void **)&c,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

values = sizeof(double) * 24;
adapt_model.type = CAM_HUNT;
adapt_model.nvars = 24;


if((adapt_model.val = (double *) malloc(values))== NULL)
{
 fprintf(stdout,  "Malloc failed\n");
 exit;
}


adapt_model.val[0] = *c;
adapt_model.val[1] = *(c+1);
adapt_model.val[2] = *(c+2);
adapt_model.val[3] = *(c+3);
adapt_model.val[4] = *(c+4);
adapt_model.val[5] = *(c+5);
adapt_model.val[6] = *(c+6);
adapt_model.val[7] = *(c+7);
adapt_model.val[8] = *(c+8);
adapt_model.val[9] = *(c+9);
adapt_model.val[10] = *(c+10);
adapt_model.val[11] = *(c+11);
adapt_model.val[12] = *(c+12);
adapt_model.val[13] = *(c+13);
adapt_model.val[14] = *(c+14);
adapt_model.val[15] = *(c+15);
adapt_model.val[16] = *(c+16);
adapt_model.val[17] = *(c+17);
adapt_model.val[18] = *(c+18);
adapt_model.val[19] = *(c+19);
adapt_model.val[20] = *(c+20);
adapt_model.val[21] = *(c+21);
adapt_model.val[22] = *(c+22);
adapt_model.val[23] = *(c+23);



*adapt_model_ptr = adapt_model;



  if(Xn == NULL || Yn == NULL || Zn == NULL)
   {
          return;
   }
   else
   {
```

```
            cxInWdgtDblSet("Renorm White Point X", Xn);
            cxInWdgtDblSet("Renorm White Point Y", Yn);
            cxInWdgtDblSet("Renorm White Point Z", Zn);
    }


        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {
                JCh[0] = (float) a[0];
                JCh[1] = (float) a[1];
                JCh[2] = (float) a[2];

                return_val = fHunt93_XYZ(CP_DOUBLE, JCh, XYZ, adapt_model_ptr);

                if(return_val == 1)
                {
                b[0] = (double) XYZ[0] / Xn;
                b[1] = (double) XYZ[1] / Yn;
                b[2] = (double) XYZ[2] / Zn;
                }
                else
                {
                fprintf(stdout, "Error condition for pixel %d, %d\n", i, j);
                fprintf(stdout, "Setting values to 0.0,  0.0,  0.0\n");
                b[0] = 0.0;
                b[1] = 0.0;
                b[2] = 0.0;
                }



                a += 3;
                b += 3;

                }
        }


}
```

fXYZ_Hunt93.c ***Hunt93_to_XYZ_Norm***

```
/*************************************************************************/
/*                                                                       */
/*  file: fXYZ_Hunt93.c */
/*  description:                                                         */
/*                                                                       */
/*  author: Mike Stokes,                                                 */
/*       Date: 1/27/93 */
```

```
/*          Mike Stokes (mdspci@judd.cis.rit.edu)          */
/*          Munsell Color Science Laboratory (MCSL)        */
/*          Center for Imaging Science                     */
/*          Rochester Institute of Technology              */
/*          Rochester, NY  14623                           */
/*                                                         */
/*  Repaired: Mark D. Fairchild 4/28/93                    */
/*                                                         */
/***********************************************************************/

#include "crs.h"

#define PI 3.14159265

extern int errno;

int fXYZ_Hunt93(comp_type, XYZ, JCh, adapt_model)
int comp_type;
float XYZ[3];
float JCh[3];
xfcn_ptr adapt_model;
{
  float LAS, Nc, Ncb, Nbb, rhoW, gammaW, betaW;
  float FL, Frho, Fgamma, Fbeta, rhoD, gammaD, betaD, Brho, Bgamma, Bbeta;
  float rhoA, gammaA, betaA, AA, C1, C2, C3, t, tprime, es;
  float Ft, MYB, MRG, M, FLS, BS, AS, A, N1, N2, QW, z, YBYW;
  float rho, gamma, beta, s, Q, hs;
  float tmpBS, tmp1, tmp2, tmp3;

 switch (comp_type) {
  case CP_CHAR :
   crs_warn2("fXYZ_Hunt93 : char computations are not yet implemented");
   return(NOT_OK);
   break;
  case CP_SHORT :
   crs_warn2("fXYZ_Hunt93 : short computations are not yet implemented");
   return(NOT_OK);
   break;
  case CP_LONG :
   crs_warn2("fXYZ_Hunt93 : long computations are not yet implemented");
   return(NOT_OK);
   break;
  case CP_DOUBLE :
  case CP_FLOAT :
   if (XYZ[0] == 0 && XYZ[1] == 0 && XYZ[2] == 0) {
        JCh[0] = JCh[1] = JCh[2] = 0.0;
        return(OK);
   }

/* Rename passed variables to make code readable */
  Brho = (float) adapt_model->val[0];
  Bgamma = (float) adapt_model->val[1];
  Bbeta = (float) adapt_model->val[2];
  FL = (float) adapt_model->val[3];
  Frho = (float) adapt_model->val[4];
```

```
  Fgamma = (float) adapt_model->val[5];
  Fbeta = (float) adapt_model->val[6];
  rhoW = (float) adapt_model->val[7];
  gammaW = (float) adapt_model->val[8];
  betaW = (float) adapt_model->val[9];
  rhoD = (float) adapt_model->val[10];
  gammaD = (float) adapt_model->val[11];
  betaD = (float) adapt_model->val[12];
  Nc = (float) adapt_model->val[13];
  Ncb = (float) adapt_model->val[14];
  Ft = (float) adapt_model->val[15];
  LAS = (float) adapt_model->val[16];
  FLS = (float) adapt_model->val[17];
  Nbb = (float) adapt_model->val[18];
  N1 = (float) adapt_model->val[19];
  N2 = (float) adapt_model->val[20];
  QW = (float) adapt_model->val[21];
  z = (float) adapt_model->val[22];
  YBYW = (float) adapt_model->val[23];



/* Calculate rho, gamma and beta */

  rho = 0.38971*XYZ[0] + 0.68898*XYZ[1] - 0.07868*XYZ[2];
  gamma = -0.22981*XYZ[0] + 1.18340*XYZ[1] + 0.04641*XYZ[2];
  beta = 1.0*XYZ[2];




/* Calculate rhoA, gammaA, betaA */
  if (FL*Frho*rho/rhoW <= 0) {
    tmp1 = 0.0;
  }
  else {
    tmp1 = pow((FL*Frho*rho/rhoW), 0.73);

  }

  if (FL*Fgamma*gamma/gammaW <= 0) {
    tmp2 = 0.0;
  }
  else {
    tmp2 = pow((FL*Fgamma*gamma/gammaW),0.73);

  }


  if (FL*Fbeta*beta/betaW <= 0) {
    tmp3 = 0.0;
  }
  else {
    tmp3 = pow((FL*Fbeta*beta/betaW),0.73);
```

```
  }

  rhoA = Brho*(40.0 * (tmp1 / (tmp1+2.0) ) + rhoD) + 1.0;
  gammaA = Bgamma*(40.0*(tmp2 / (tmp2+2.0) ) + gammaD) + 1.0;
  betaA = Bbeta*(40.0*(tmp3 / (tmp3 + 2.0) ) + betaD) + 1.0;


/* Calculate AA */
  AA = 2.0*rhoA + gammaA + 0.05*betaA - 3.05 + 1.0;

/* Calculate C1, C2, C3 */
  C1 = rhoA - gammaA;
  C2 = gammaA - betaA;
  C3 = betaA - rhoA;
fprintf(stdout, "C1=%f C2=%f C3=%f\n", C1, C2, C3);
/* Calculate t and tprime */
  t = 0.5*(C2 - C3)/4.5;
  tprime = C1 - (C2/11.0);

fprintf(stdout, "t = %f tprime = %f\n", t, tprime);

/* Calculate hs */
  errno = 0;
  hs = atan2f(t,tprime)*(180/PI);
  if (errno) {
    printf("XYZ %f %f %f\n", XYZ[0], XYZ[1], XYZ[2]);
    printf("errno = %d, t = %f, tprime = %f\n", errno, t, tprime);
    crs_error2("fXYZ_Hunt93 : atan2 error");
    return(NOT_OK);
  }
  if(hs<0) hs = 360 + hs;
  JCh[2] = hs;

/* calculate es */
  if(hs >= 0.0 && hs < 20.14)
     es = 1.2 - 0.4*(hs+122.47)/142.61;

  else if(hs >= 20.14 && hs < 90.0)
    es = 0.8 - 0.1*(hs-20.14)/69.86;

  else if(hs >= 90.0 && hs < 164.25)
    es = 0.7 + 0.3*(hs-90.0)/74.25;

  else if(hs >= 164.25 && hs < 237.53)
    es = 1.0 + 0.2*(hs-164.25)/73.28;

  else
    es = 1.2 - 0.4*(hs-237.53)/142.61;

/* Calcualte MYB, MRG, and M */
  MYB = 100.0*t*es*(10.0/13.0)*Nc*Ncb*Ft;
  MRG = 100.0*tprime*es*(10.0/13.0)*Nc*Ncb;
  M = pow((MYB*MYB + MRG*MRG),0.5);

/* Calculate s */
```

```
    s = (50.0 * M) / (rhoA + gammaA + betaA);

  /* Calculate BS */
    tmpBS = pow(((5.0*LAS/2.26) * (XYZ[1]/100.0)), 0.3);

    BS = 0.5 / (1.0+0.3*tmpBS) + 0.5/(1.0+5.0*(5.0*LAS/2.26));

  /* Calculate AS */
    tmp1 = pow((FLS*((double)XYZ[1]/100.0)),0.73);

    AS = BS*3.05 * (40.0*(tmp1 / (tmp1 + 2.0))) + 0.3;

  /* Calculate A */
    A = Nbb*(AA - 1.0 + AS - 0.3 + 1.0440365);

  /* Calculate Q */
    tmpBS = 7.0*(A + M/100.0);
     tmp1 = pow(tmpBS, 0.6);

    Q = tmp1 * N1 - N2;

  /* Calculate J */
    if (Q >= 0.0) {
      tmp2 = pow((Q/QW), z);
      JCh[0] = 100.0 * tmp2;
    }
    else {
      JCh[0] = 0.0;
    }

  /* Calculate Cb */
    tmp1 = pow(s, 0.69);

    if (Q >= 0.0) {
      tmp2 = pow((Q/QW),YBYW);
     }
     else {
       tmp2 = 0.0;
     }

    tmp3 = pow(0.29, YBYW);

    JCh[1] =  2.44*tmp1*tmp2*(1.64 - tmp3);

      break;
  }
  return(OK);
}
```

fHunt93_XYZ.c ***Hunt93_to_XYZ_Norm***

```
/*********************************************************************/
/*                                                                   */
/*  file: fHunt93_XYZ.c */
```

```
/* description:                                                    */
/*                                                                 */
/* author: Mike Stokes,                                            */
/*       Date: 1/27/93 */
/*            Mike Stokes (mdspci@judd.cis.rit.edu)                */
/*            Munsell Color Science Laboratory (MCSL)              */
/*            Center for Imaging Science                           */
/*            Rochester Institute of Technology                    */
/*            Rochester, NY  14623                                 */
/*                                                                 */
/*  Repaired: Mark D. Fairchild 4/28/93                            */
/*                      6/11/93                                    */
/*                                                                 */
/*******************************************************************/

#include "crs.h"
#include <math.h>

extern int errno;

int fHunt93_XYZ(comp_type, JCh, XYZ, adapt_model)
int comp_type;
float JCh[3];
float XYZ[3];
xfcn_ptr adapt_model;
{
  static flag = OK;
  static float xyz[4];

  switch (comp_type) {
   case CP_CHAR :
    crs_warn2("fHunt93_XYZ : char computations are not yet implemented");
    return(NOT_OK);
    break;
   case CP_SHORT :
    crs_warn2("fHunt93_XYZ : short computations are not yet implemented");
    return(NOT_OK);
    break;
   case CP_LONG :
    crs_warn2("fHunt93_XYZ : long computations are not yet implemented");
    return(NOT_OK);
    break;
   case CP_DOUBLE :
   case CP_FLOAT :
    if (flag == OK)
/*      xyz[0] = xyz[1] = xyz[2] = xyz[3] = 25.0;  */

          xyz[0] = 25.0;
       xyz[1] = 26.0;
       xyz[2] = 27.0;
       xyz[3] = 25.0;

         if (JCh[0] == 0) {
                 XYZ[0] = 0.0;
                 XYZ[1] = 0.0;
```

```
                XYZ[2] = 0.0;
                return(OK);
                }

    if (!H93mnewt(comp_type, 100, xyz, 3, 0.15, 0.15, JCh, adapt_model)) {
        printf("JCh %f %f %f -> XYZ %f %f %f\n",
          JCh[0], JCh[1], JCh[2], xyz[1], xyz[2], xyz[3]);
        crs_warn2("fHunt93_XYZ : H93mnewt");
        return(NOT_OK);
    }

    XYZ[0] = xyz[1];
    XYZ[1] = xyz[2];
    XYZ[2] = xyz[3];

    break;
  }
  return(OK);
}
```

XYZ2RLAB.c *XYZ_Norm_to_RLAB*

```
/*

  XYZ2RLAB.c

  This file reads in a lattice of Xn/Yn/Zn, renormalizes the data to
  absolute XYZ's and then runs the data through the RLAB 95 model to
  produce Lr, ar, br.

  b[0] = Lr;
  b[1] = ar;
  b[2] = br;

  Based on F95PreCalc.c by Mark Fairchild.  See below

*/


/*************************************************************************/
/*                                                                       */
/* file: F95PreCalc.c                                                    */
/* description: Precalculate matrices for F91_XYZ and XYZ_F91            */
/*          Modified model includes discounting factor D                 */
/*          and eliminates the deliterious C matric                      */
/*          with new 1995 normalizations                                 */
/*                                                                       */
/* author:   Mark Fairchild                                              */
/*       Date: 1989-1993-1995                                            */
/*          Munsell Color Science Laboratory (MCSL)                      */
/*          Center for Imaging Science                                   */
/*          Rochester Institute of Technology                            */
/*          Rochester, NY  14623                                         */
/*                                                                       */
```

```
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


#define LX 0.3897
#define LY 0.6890
#define LZ -0.0787
#define MX -0.2298
#define MY 1.1834
#define MZ 0.0464
#define SX 0.0000
#define SY 0.0000
#define SZ 1.0000

#define XL 1.9102
#define XM -1.1122
#define XS 0.2019
#define YL 0.3709
#define YM 0.6291
#define YS 0.0000
#define ZL 0.0000
#define ZM 0.0000
#define ZS 1.0000

#define R1 1.9569
#define R2 -1.1882
#define R3 0.2313
#define R4 0.3612
#define R5 0.6388
#define R6 0.0000
#define R7 0.0000
#define R8 0.0000
#define R9 1.0000

#define LE 1.0000
#define ME 1.0000
#define SE 1.0000

#define nu 0.3333

#define SOFTCOPY 0
#define HARDCOPY 1

void XYZ_to_RLAB(cxLattice *in, cxLattice **out, double Xnr, double Ynr,
        double Znr, double Xn, double Yn, double Zn, double Yabsn,
                double factor,  double which_sigma, double adjust_sigma)
{
```

```
double *a;
double *b;
cxErrorCode err;
int i, j;
double al, am, as, c;
double Ytonu, LMSn[3], lmsE[3], pl, pm, ps, iCon, iCoff;
double forMatrix[9], tmpforMatrix[9], finalMatrix[9];
int cogswitch;
double tempX, tempY, tempZ, Xr, Yr, Zr ;
double sigma;



        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,                       /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
        fprintf(stdout, "I have made it here!\n");

        if(Xnr == NULL || Ynr == NULL || Znr == NULL)
        {
          return;
        }
        if(Yabsn == NULL || factor == NULL)
        {
          return;
        }

        /* Setup custom sigma dial range */

        cxInWdgtDblMinMaxSet("Custom sigma", 1.0, 4.0);

        /* Decide which sigma to use       */

        switch(which_sigma)
        {
          case 0:

          cxInWdgtDisable("Custom sigma");

          sigma = (1.0 / 2.3);
          break;

          case 1:
```

```
            cxInWdgtDisable("Custom sigma");

            sigma = (1.0 / 2.9);
            break;

            case 2:

            cxInWdgtDisable("Custom sigma");

            sigma = (1.0 / 3.5);
            break;

            case 3:

            cxInWdgtEnable("Custom sigma");
            sigma = (1.0 / adjust_sigma);

          }



/* convert XYZ to LMS */
  LMSn[0] = LX * (double) Xnr + LY * (double) Ynr + LZ * (double) Znr;
  LMSn[1] = MX * (double) Xnr + MY * (double) Ynr + MZ * (double) Znr;
  LMSn[2] = SX * (double) Xnr + SY * (double) Ynr + SZ * (double) Znr;

/* calculate relative chromaticities */
  lmsE[0] = (3 * (LMSn[0]/LE))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
  lmsE[1] = (3 * (LMSn[1]/ME))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
  lmsE[2] = (3 * (LMSn[2]/SE))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));

/* calculate pl, pm, ps */
  Ytonu = pow((double) Yabsn,nu);

  pl = (1.0 + Ytonu + lmsE[0])/(1.0 + Ytonu + (1.0/lmsE[0]));
  pm = (1.0 + Ytonu + lmsE[1])/(1.0 + Ytonu + (1.0/lmsE[1]));
  ps = (1.0 + Ytonu + lmsE[2])/(1.0 + Ytonu + (1.0/lmsE[2]));

/* apply discounting factor to pl, pm, ps */

  pl = pl + (double) factor *(1.0-pl);
  pm = pm + (double) factor *(1.0-pm);
  ps = ps + (double) factor *(1.0-ps);

/* calculate al, am, as */
  al = pl/LMSn[0];
  am = pm/LMSn[1];
  as = ps/LMSn[2];

/* calculate c */
/* since c is no longer used, it is set to zero */
  c = 0.0;
```

```
/* calculate AM matrix */
  tmpforMatrix[0] = al * LX;
  tmpforMatrix[1] = al * LY;
  tmpforMatrix[2] = al * LZ;
  tmpforMatrix[3] = am * MX;
  tmpforMatrix[4] = am * MY;
  tmpforMatrix[5] = am * MZ;
  tmpforMatrix[6] = as * SX;
  tmpforMatrix[7] = as * SY;
  tmpforMatrix[8] = as * SZ;

/* calculate CAM (forMatrix) matrix */
  forMatrix[0] = tmpforMatrix[0] + c * tmpforMatrix[3] + c * tmpforMatrix[6];
  forMatrix[1] = tmpforMatrix[1] + c * tmpforMatrix[4] + c * tmpforMatrix[7];
  forMatrix[2] = tmpforMatrix[2] + c * tmpforMatrix[5] + c * tmpforMatrix[8];
  forMatrix[3] = c * tmpforMatrix[0] + tmpforMatrix[3] + c * tmpforMatrix[6];
  forMatrix[4] = c * tmpforMatrix[1] + tmpforMatrix[4] + c * tmpforMatrix[7];
  forMatrix[5] = c * tmpforMatrix[2] + tmpforMatrix[5] + c * tmpforMatrix[8];
  forMatrix[6] = c * tmpforMatrix[0] + c * tmpforMatrix[3] + tmpforMatrix[6];
  forMatrix[7] = c * tmpforMatrix[1] + c * tmpforMatrix[4] + tmpforMatrix[7];
  forMatrix[8] = c * tmpforMatrix[2] + c * tmpforMatrix[5] + tmpforMatrix[8];

/* calculate RAM matrix   (finalMatrix)  */



  finalMatrix[0] = ((R1 * forMatrix[0]) + (R2 * forMatrix[3]) + (R3 * forMatrix[6]));
  finalMatrix[1] = ((R1 * forMatrix[1]) + (R2 * forMatrix[4]) + (R3 * forMatrix[7]));
  finalMatrix[2] = ((R1 * forMatrix[2]) + (R2 * forMatrix[5]) + (R3 * forMatrix[8]));
  finalMatrix[3] = ((R4 * forMatrix[0]) + (R5 * forMatrix[3]) + (R6 * forMatrix[6]));
  finalMatrix[4] = ((R4 * forMatrix[1]) + (R5 * forMatrix[4]) + (R6 * forMatrix[7]));
  finalMatrix[5] = ((R4 * forMatrix[2]) + (R5 * forMatrix[5]) + (R6 * forMatrix[8]));
  finalMatrix[6] = ((R7 * forMatrix[0]) + (R8 * forMatrix[3]) + (R9 * forMatrix[6]));
  finalMatrix[7] = ((R7 * forMatrix[1]) + (R8 * forMatrix[4]) + (R9 * forMatrix[7]));
  finalMatrix[8] = ((R7 * forMatrix[2]) + (R8 * forMatrix[5]) + (R9 * forMatrix[8]));

 for(i = 0; i < 9; i++)
 {

 fprintf(stdout,  "The AM matrix value at position %d is %14.7lf\n",  i,  forMatrix[i]);

  }

 for(i = 0; i < 9; i++)
 {

 fprintf(stdout,  "The RAM matrix value at position %d is %14.7lf\n",  i,  finalMatrix[i]);

 }



        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
```

```
                for ( j = 0; j < in->dims[0]; j++ )
                {

                tempX = a[0] * Xn;
                tempY = a[1] * Yn;
                tempZ = a[2] * Zn;

                Xr = ((finalMatrix[0] * tempX) + (finalMatrix[1] * tempY) +
                     (finalMatrix[2] * tempZ));

                Yr = ((finalMatrix[3] * tempX) + (finalMatrix[4] * tempY) +
                     (finalMatrix[5] * tempZ));

                Zr = ((finalMatrix[6] * tempX) + (finalMatrix[7] * tempY) +
                     (finalMatrix[8] * tempZ));



                b[0] = 100.0 * pow(Yr,sigma);

                b[1] = (430.0 * (pow(Xr, sigma) - pow(Yr,  sigma)));

                b[2] = (170.0 * (pow(Yr, sigma) - pow(Zr,  sigma)));


                a += 3;
                b += 3;

                }

        }

}
```

RLAB2XYZ.c ***RLAB_to_XYZ_Norm***

```
/*

  RLAB2XYZ.c

  This function takes in RLAB Lr, ar, br values and converts them back
  to Xnorm,  Ynorm,  and Znorm tristumulus values using the RLAB 95
  model.

  b[0] = Xn;
  b[1] = Yn;
  b[2] = Zn;

  Based on F95PreCalc.c by Mark Fairchild.  See below

*/
```

```
/*****************************************************************/
/*                                                               */
/*  file: F95PreCalc.c                                           */
/*  description: Precalculate matrices for F91_XYZ and XYZ_F91   */
/*           Modified model includes discounting factor D        */
/*           and eliminates the deliterious C matric             */
/*           with new 1995 normalizations                        */
/*                                                               */
/*  author:   Mark Fairchild                                     */
/*       Date: 1989-1993-1995                                    */
/*           Munsell Color Science Laboratory (MCSL)             */
/*           Center for Imaging Science                          */
/*           Rochester Institute of Technology                   */
/*           Rochester, NY  14623                                */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


#define LX 0.3897
#define LY 0.6890
#define LZ -0.0787
#define MX -0.2298
#define MY 1.1834
#define MZ 0.0464
#define SX 0.0000
#define SY 0.0000
#define SZ 1.0000

#define XL 1.9102
#define XM -1.1122
#define XS 0.2019
#define YL 0.3709
#define YM 0.6291
#define YS 0.0000
#define ZL 0.0000
#define ZM 0.0000
#define ZS 1.0000

#define R1 0.3804
#define R2 0.7076
#define R3 -0.0880
#define R4 -0.2151
#define R5 1.1653
#define R6 0.0480
#define R7 0.0000
#define R8 0.0000
#define R9 1.0000
```

```
#define LE 1.0000
#define ME 1.0000
#define SE 1.0000

#define nu 0.3333

#define SOFTCOPY 0
#define HARDCOPY 1

void RLAB_to_XYZ_Norm(cxLattice *in, cxLattice **out, double Xnr, double Ynr,
        double Znr, double Xn, double Yn, double Zn, double Yabsn,
                double factor,  double which_sigma, double adjust_sigma)
{


  double *a;
  double *b;
  cxErrorCode err;
  int i,  j;
  double al, am, as, c;
  double Ytonu, LMSn[3], lmsE[3], pl, pm, ps, iCon, iCoff;
  double invMatrix[9], tmpinvMatrix[9], finalMatrix[9];
  int cogswitch;
  double tempX,  tempY,  tempZ, Xr,  Yr,  Zr;
  double sigma,  invSigma;
  double temp1,  temp2;



        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,                       /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
        fprintf(stdout,  "I have made it here!\n");

        if(Xnr == NULL || Ynr == NULL || Znr == NULL || Xn == NULL || Yn == NULL
          || Zn == NULL)
        {
           return;
        }
        else
        {
           cxInWdgtDblSet("White Point X",  Xnr);
           cxInWdgtDblSet("White Point Y",  Ynr);
```

```
    cxInWdgtDblSet("White Point Z", Znr);
    cxInWdgtDblSet("Adapting WP X", Xn);
    cxInWdgtDblSet("Adapting WP Y", Yn);
    cxInWdgtDblSet("Adapting WP Z", Zn);
}
if(Yabsn == NULL)
{
    return;
}
else
{
    cxInWdgtDblSet("Adapting Luminance", Yabsn);
}

/* Setup custom sigma dial range */

cxInWdgtDblMinMaxSet("Custom sigma", 1.0, 4.0);

/* Decide which sigma to use        */

switch(which_sigma)
{
    case 0:

    cxInWdgtDisable("Custom sigma");

    sigma = (1.0 / 2.3);
    invSigma = 2.3;
    break;

    case 1:

    cxInWdgtDisable("Custom sigma");

    sigma = (1.0 / 2.9);
    invSigma = 2.9;
    break;

    case 2:

    cxInWdgtDisable("Custom sigma");

    sigma = (1.0 / 3.5);
    invSigma = 3.5;
    break;

    case 3:

    cxInWdgtEnable("Custom sigma");
    sigma = (1.0 / adjust_sigma);
    invSigma = adjust_sigma;
    break;


}
```

```
/* convert XYZ to LMS */
  LMSn[0] = LX * (double) Xn + LY * (double) Yn + LZ * (double) Zn;
  LMSn[1] = MX * (double) Xn + MY * (double) Yn + MZ * (double) Zn;
  LMSn[2] = SX * (double) Xn + SY * (double) Yn + SZ * (double) Zn;

/* calculate relative chromaticities */
  lmsE[0] = (3.0 * (LMSn[0]/LE))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
  lmsE[1] = (3.0 * (LMSn[1]/ME))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
  lmsE[2] = (3.0 * (LMSn[2]/SE))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));

/* calculate pl, pm, ps */
  Ytonu = pow((double) Yabsn,nu);

  pl = (1.0 + Ytonu + lmsE[0])/(1.0 + Ytonu + (1.0/lmsE[0]));
  pm = (1.0 + Ytonu + lmsE[1])/(1.0 + Ytonu + (1.0/lmsE[1]));
  ps = (1.0 + Ytonu + lmsE[2])/(1.0 + Ytonu + (1.0/lmsE[2]));

/* apply discounting factor to pl, pm, ps */

  pl = pl + (double) factor *(1.0-pl);
  pm = pm + (double) factor *(1.0-pm);
  ps = ps + (double) factor *(1.0-ps);

/* calculate al, am, as */
  al = pl/LMSn[0];
  am = pm/LMSn[1];
  as = ps/LMSn[2];

/* calculate c */
/* since c is no longer used, it is set to zero */
  c = 0.0;

  iCon = 1.0;
  iCoff = 0.0;


/* calculate invAinvC matrix */
  tmpinvMatrix[0] = (1.0/al) * iCon;
  tmpinvMatrix[1] = (1.0/al) * iCoff;
  tmpinvMatrix[2] = (1.0/al) * iCoff;
  tmpinvMatrix[3] = (1.0/am) * iCoff;
  tmpinvMatrix[4] = (1.0/am) * iCon;
  tmpinvMatrix[5] = (1.0/am) * iCoff;
  tmpinvMatrix[6] = (1.0/as) * iCoff;
  tmpinvMatrix[7] = (1.0/as) * iCoff;
  tmpinvMatrix[8] = (1.0/as) * iCon;

/* calculate invMinvAinvC (invMatrix) matrix */
  invMatrix[0] = XL * tmpinvMatrix[0] + XM * tmpinvMatrix[3] + XS * tmpinvMatrix[6];
  invMatrix[1] = XL * tmpinvMatrix[1] + XM * tmpinvMatrix[4] + XS * tmpinvMatrix[7];
  invMatrix[2] = XL * tmpinvMatrix[2] + XM * tmpinvMatrix[5] + XS * tmpinvMatrix[8];
```

```
invMatrix[3] = YL * tmpinvMatrix[0] + YM * tmpinvMatrix[3] + YS * tmpinvMatrix[6];
invMatrix[4] = YL * tmpinvMatrix[1] + YM * tmpinvMatrix[4] + YS * tmpinvMatrix[7];
invMatrix[5] = YL * tmpinvMatrix[2] + YM * tmpinvMatrix[5] + YS * tmpinvMatrix[8];
invMatrix[6] = ZL * tmpinvMatrix[0] + ZM * tmpinvMatrix[3] + ZS * tmpinvMatrix[6];
invMatrix[7] = ZL * tmpinvMatrix[1] + ZM * tmpinvMatrix[4] + ZS * tmpinvMatrix[7];
invMatrix[8] = ZL * tmpinvMatrix[2] + ZM * tmpinvMatrix[5] + ZS * tmpinvMatrix[8];


/* calculate RAM matrix   (finalMatrix)  */



finalMatrix[0] = ((R1 * invMatrix[0]) + (R2 * invMatrix[3]) + (R3 * invMatrix[6]));
finalMatrix[1] = ((R1 * invMatrix[1]) + (R2 * invMatrix[4]) + (R3 * invMatrix[7]));
finalMatrix[2] = ((R1 * invMatrix[2]) + (R2 * invMatrix[5]) + (R3 * invMatrix[8]));
finalMatrix[3] = ((R4 * invMatrix[0]) + (R5 * invMatrix[3]) + (R6 * invMatrix[6]));
finalMatrix[4] = ((R4 * invMatrix[1]) + (R5 * invMatrix[4]) + (R6 * invMatrix[7]));
finalMatrix[5] = ((R4 * invMatrix[2]) + (R5 * invMatrix[5]) + (R6 * invMatrix[8]));
finalMatrix[6] = ((R7 * invMatrix[0]) + (R8 * invMatrix[3]) + (R9 * invMatrix[6]));
finalMatrix[7] = ((R7 * invMatrix[1]) + (R8 * invMatrix[4]) + (R9 * invMatrix[7]));
finalMatrix[8] = ((R7 * invMatrix[2]) + (R8 * invMatrix[5]) + (R9 * invMatrix[8]));

for(i = 0; i < 9; i++)
{

fprintf(stdout, "The AM^-1 matrix value at position %d is %14.7lf\n", i, invMatrix[i]);

 }

for(i = 0; i < 9; i++)
{

fprintf(stdout, "The RAM^-1 matrix value at position %d is %14.7lf\n", i, finalMatrix[i]);

}



        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {

                Yr = pow((a[0] / 100.0), invSigma);

                temp1 = pow(Yr, sigma);
                temp2 = a[1] / 430.0;


                Xr = pow((temp2 + temp1), invSigma);
                Zr = pow((pow(Yr, sigma) - (a[2] / 170.0)), invSigma);
```

```
                tempX = ((finalMatrix[0] * Xr) + (finalMatrix[1] * Yr) +
                    (finalMatrix[2] * Zr));

                tempY = ((finalMatrix[3] * Xr) + (finalMatrix[4] * Yr) +
                    (finalMatrix[5] * Zr));

                tempZ = ((finalMatrix[6] * Xr) + (finalMatrix[7] * Yr) +
                    (finalMatrix[8] * Zr));



                b[0] = tempX / Xnr;

                b[1] = tempY / Ynr;

                b[2] = tempZ / Znr;


                a += 3;
                b += 3;

                }

        }

}
```

XYZ2RLAB_var.c ***XYZ_Norm_to_RLAB_Variable***

```
/*

  XYZ2RLAB.c

  This file reads in a lattice of Xn/Yn/Zn,  renormalizes the data to
  absolute XYZ's and then runs the data through the RLAB 95 model to
  produce Lr,  ar,  br.

  Modified 7/9/96 to add variable exponents on Lr as ar/br.

  Chris Hauf

  b[0] = Lr;
  b[1] = ar;
  b[2] = br;

  Based on F95PreCalc.c by Mark Fairchild.  See below

*/


/*****************************************************************/
/*                                                               */
/*  file: F95PreCalc.c                                           */
```

```
/* description: Precalculate matrices for F91_XYZ and XYZ_F91      */
/*          Modified model includes discounting factor D       */
/*          and eliminates the deliterious C matric          */
/*          with new 1995 normalizations                    */
/*                                                          */
/* author:   Mark Fairchild                                          */
/*       Date: 1989-1993-1995                                        */
/*          Munsell Color Science Laboratory (MCSL)          */
/*          Center for Imaging Science                      */
/*          Rochester Institute of Technology               */
/*          Rochester, NY  14623                            */
/*                                                          */
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


#define LX 0.3897
#define LY 0.6890
#define LZ -0.0787
#define MX -0.2298
#define MY 1.1834
#define MZ 0.0464
#define SX 0.0000
#define SY 0.0000
#define SZ 1.0000

#define XL 1.9102
#define XM -1.1122
#define XS 0.2019
#define YL 0.3709
#define YM 0.6291
#define YS 0.0000
#define ZL 0.0000
#define ZM 0.0000
#define ZS 1.0000

#define R1 1.9569
#define R2 -1.1882
#define R3 0.2313
#define R4 0.3612
#define R5 0.6388
#define R6 0.0000
#define R7 0.0000
#define R8 0.0000
#define R9 1.0000

#define LE 1.0000
#define ME 1.0000
#define SE 1.0000
```

```
#define nu 0.3333

#define SOFTCOPY 0
#define HARDCOPY 1

void XYZ_to_RLAB_Variable(cxLattice *in, cxLattice **out, double Xnr, double Ynr,
        double Znr, double Xn, double Yn, double Zn, double Yabsn,
                double factor,  int variable_sigma, int which_sigma_lr,
                int which_sigma_ar_br,
                double adjust_sigma, double adjust_sigma_ar_br)
{


  double *a;
  double *b;
  cxErrorCode err;
  int i,  j;
  double al, am, as, c;
  double Ytonu, LMSn[3], lmsE[3], pl, pm, ps, iCon, iCoff;
  double forMatrix[9], tmpforMatrix[9], finalMatrix[9];
  int cogswitch;
  double tempX,  tempY,  tempZ, Xr,  Yr,  Zr ;
  double sigma,  sigma_ar_br;




        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,                       /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
        fprintf(stdout,  "I have made it here!\n");

        if(Xnr == NULL || Ynr == NULL || Znr == NULL || Xn == NULL || Yn == NULL
      || Zn == NULL)
        {
          return;
        }
        else
        {
          cxInWdgtDblSet("White Point X",  Xnr);
          cxInWdgtDblSet("White Point Y",  Ynr);
          cxInWdgtDblSet("White Point Z",  Znr);
          cxInWdgtDblSet("Adapting WP X",  Xn);
```

```
    cxInWdgtDblSet("Adapting WP Y", Yn);
    cxInWdgtDblSet("Adapting WP Z", Zn);
  }
  if(Yabsn == NULL)
  {
    return;
  }
  else
  {
    cxInWdgtDblSet("Adapting Luminance", Yabsn);
  }


  /*Decide whether want same sigma for all channels or variable sigmas
   * for luma/chromas
   */

  if(variable_sigma == 0)
  {

  /*Disable & hide ar/br sigma controls */

  cxInWdgtDisable("Sigma ar/br");
cxInWdgtDisable("Custom sigma ar/br");
  cxInWdgtHide("Custom sigma ar/br");


          /* Setup custom sigma dial range */

  cxInWdgtDblMinMaxSet("Custom sigma Lr/all", 1.0, 4.0);

  /* Decide which sigma to use        */

  switch(which_sigma_lr)
  {
    case 0:

    cxInWdgtDisable("Custom sigma Lr/all");
    cxInWdgtHide("Custom sigma Lr/all");

    sigma = (1.0 / 2.3);
    sigma_ar_br = (1.0 / 2.3);
    break;

    case 1:

    cxInWdgtDisable("Custom sigma Lr/all");
    cxInWdgtHide("Custom sigma Lr/all");

    sigma = (1.0 / 2.9);
    sigma_ar_br = (1.0 / 2.9);
    break;

    case 2:
```

```
            cxInWdgtDisable("Custom sigma Lr/all");
            cxInWdgtHide("Custom sigma Lr/all");

            sigma = (1.0 / 3.5);
            sigma_ar_br = (1.0 / 3.5);
            break;

            case 3:

            cxInWdgtShow("Custom sigma Lr/all");
            cxInWdgtEnable("Custom sigma Lr/all");
            sigma = (1.0 / adjust_sigma);
            sigma_ar_br = (1.0 / adjust_sigma);
            break;

        }
}

else if(variable_sigma == 1)
{
        /*Enable and show ar/br sigma controls */

        cxInWdgtEnable("Sigma ar/br");


        /* Setup custom sigma dial range */

        cxInWdgtDblMinMaxSet("Custom sigma Lr/all", 1.0, 4.0);
        cxInWdgtDblMinMaxSet("Custom sigma ar/br", 1.0, 4.0);

        /* Decide which Lr sigma to use     */

        switch(which_sigma_lr)
        {
          case 0:

          cxInWdgtDisable("Custom sigma Lr/all");
          cxInWdgtHide("Custom sigma Lr/all");

          sigma = (1.0 / 2.3);
          break;

          case 1:

          cxInWdgtDisable("Custom sigma Lr/all");
          cxInWdgtHide("Custom sigma Lr/all");

          sigma = (1.0 / 2.9);
          break;

          case 2:

          cxInWdgtDisable("Custom sigma Lr/all");
          cxInWdgtHide("Custom sigma Lr/all");
```

```
            sigma = (1.0 / 3.5);
            break;

            case 3:

            cxInWdgtShow("Custom sigma Lr/all");
            cxInWdgtEnable("Custom sigma Lr/all");
            sigma = (1.0 / adjust_sigma);
            break;

        }

        switch(which_sigma_ar_br)
        {
            case 0:

            cxInWdgtDisable("Custom sigma ar/br");
            cxInWdgtHide("Custom sigma ar/br");

            sigma_ar_br = (1.0 / 2.3);
            break;

            case 1:

            cxInWdgtDisable("Custom sigma ar/br");
            cxInWdgtHide("Custom sigma ar/br");

            sigma_ar_br = (1.0 / 2.9);
            break;

            case 2:

            cxInWdgtDisable("Custom sigma ar/br");
            cxInWdgtHide("Custom sigma ar/br");

            sigma_ar_br = (1.0 / 3.5);
            break;

            case 3:

            cxInWdgtShow("Custom sigma ar/br");
            cxInWdgtEnable("Custom sigma ar/br");
            sigma_ar_br = (1.0 / adjust_sigma_ar_br);
            break;

        }

    }


/* convert XYZ to LMS */
  LMSn[0] = LX * (double) Xn + LY * (double) Yn + LZ * (double) Zn;
  LMSn[1] = MX * (double) Xn + MY * (double) Yn + MZ * (double) Zn;
  LMSn[2] = SX * (double) Xn + SY * (double) Yn + SZ * (double) Zn;
```

```
/* calculate relative chromaticities */
  lmsE[0] = (3.0 * (LMSn[0]/LE))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
  lmsE[1] = (3.0 * (LMSn[1]/ME))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
  lmsE[2] = (3.0 * (LMSn[2]/SE))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));

/* calculate pl, pm, ps */
  Ytonu = pow((double) Yabsn,nu);

  pl = (1.0 + Ytonu + lmsE[0])/(1.0 + Ytonu + (1.0/lmsE[0]));
  pm = (1.0 + Ytonu + lmsE[1])/(1.0 + Ytonu + (1.0/lmsE[1]));
  ps = (1.0 + Ytonu + lmsE[2])/(1.0 + Ytonu + (1.0/lmsE[2]));

/* apply discounting factor to pl, pm, ps */

  pl = pl + (double) factor *(1.0-pl);
  pm = pm + (double) factor *(1.0-pm);
  ps = ps + (double) factor *(1.0-ps);

/* calculate al, am, as */
  al = pl/LMSn[0];
  am = pm/LMSn[1];
  as = ps/LMSn[2];

/* calculate c */
/* since c is no longer used, it is set to zero */
  c = 0.0;

/* calculate AM matrix */
  tmpforMatrix[0] = al * LX;
  tmpforMatrix[1] = al * LY;
  tmpforMatrix[2] = al * LZ;
  tmpforMatrix[3] = am * MX;
  tmpforMatrix[4] = am * MY;
  tmpforMatrix[5] = am * MZ;
  tmpforMatrix[6] = as * SX;
  tmpforMatrix[7] = as * SY;
  tmpforMatrix[8] = as * SZ;

/* calculate CAM (forMatrix) matrix */
  forMatrix[0] = tmpforMatrix[0] + c * tmpforMatrix[3] + c * tmpforMatrix[6];
  forMatrix[1] = tmpforMatrix[1] + c * tmpforMatrix[4] + c * tmpforMatrix[7];
  forMatrix[2] = tmpforMatrix[2] + c * tmpforMatrix[5] + c * tmpforMatrix[8];
  forMatrix[3] = c * tmpforMatrix[0] + tmpforMatrix[3] + c * tmpforMatrix[6];
  forMatrix[4] = c * tmpforMatrix[1] + tmpforMatrix[4] + c * tmpforMatrix[7];
  forMatrix[5] = c * tmpforMatrix[2] + tmpforMatrix[5] + c * tmpforMatrix[8];
  forMatrix[6] = c * tmpforMatrix[0] + c * tmpforMatrix[3] + tmpforMatrix[6];
  forMatrix[7] = c * tmpforMatrix[1] + c * tmpforMatrix[4] + tmpforMatrix[7];
  forMatrix[8] = c * tmpforMatrix[2] + c * tmpforMatrix[5] + tmpforMatrix[8];

/* calculate RAM matrix   (finalMatrix)  */



  finalMatrix[0] = ((R1 * forMatrix[0]) + (R2 * forMatrix[3]) + (R3 * forMatrix[6]));
  finalMatrix[1] = ((R1 * forMatrix[1]) + (R2 * forMatrix[4]) + (R3 * forMatrix[7]));
```

```
finalMatrix[2] = ((R1 * forMatrix[2]) + (R2 * forMatrix[5]) + (R3 * forMatrix[8]));
finalMatrix[3] = ((R4 * forMatrix[0]) + (R5 * forMatrix[3]) + (R6 * forMatrix[6]));
finalMatrix[4] = ((R4 * forMatrix[1]) + (R5 * forMatrix[4]) + (R6 * forMatrix[7]));
finalMatrix[5] = ((R4 * forMatrix[2]) + (R5 * forMatrix[5]) + (R6 * forMatrix[8]));
finalMatrix[6] = ((R7 * forMatrix[0]) + (R8 * forMatrix[3]) + (R9 * forMatrix[6]));
finalMatrix[7] = ((R7 * forMatrix[1]) + (R8 * forMatrix[4]) + (R9 * forMatrix[7]));
finalMatrix[8] = ((R7 * forMatrix[2]) + (R8 * forMatrix[5]) + (R9 * forMatrix[8]));

for(i = 0; i < 9; i++)
{

fprintf(stdout, "The AM matrix value at position %d is %14.7lf\n", i, forMatrix[i]);

 }

for(i = 0; i < 9; i++)
{

fprintf(stdout, "The RAM matrix value at position %d is %14.7lf\n", i, finalMatrix[i]);

}



        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {

                tempX = a[0] * Xnr;
                tempY = a[1] * Ynr;
                tempZ = a[2] * Znr;

                Xr = ((finalMatrix[0] * tempX) + (finalMatrix[1] * tempY) +
                     (finalMatrix[2] * tempZ));

                Yr = ((finalMatrix[3] * tempX) + (finalMatrix[4] * tempY) +
                     (finalMatrix[5] * tempZ));

                Zr = ((finalMatrix[6] * tempX) + (finalMatrix[7] * tempY) +
                     (finalMatrix[8] * tempZ));



                b[0] = 100.0 * pow(Yr,sigma);

                b[1] = (430.0 * (pow(Xr, sigma_ar_br) - pow(Yr,  sigma_ar_br)));

                b[2] = (170.0 * (pow(Yr, sigma_ar_br) - pow(Zr,  sigma_ar_br)));


                a += 3;
                b += 3;
```

```
            }
        }
}
```

RLAB2XYZ_var.c *RLAB_to_XYZ_Norm_Variable*

```
/*
   RLAB2XYZ.c

   This function takes in RLAB Lr, ar, br values and converts them back
   to Xnorm, Ynorm, and Znorm tristumulus values using the RLAB 95
   model.

   b[0] = Xn;
   b[1] = Yn;
   b[2] = Zn;

   Based on F95PreCalc.c by Mark Fairchild. See below

   Modified 10/3/96 to add variable exponents to Lr and ar/br (CRH)
*/


/*************************************************************************/
/*                                                                       */
/* file: F95PreCalc.c                                                    */
/* description: Precalculate matrices for F91_XYZ and XYZ_F91            */
/*          Modified model includes discounting factor D                 */
/*          and eliminates the deliterious C matric                      */
/*          with new 1995 normalizations                                 */
/*                                                                       */
/* author:   Mark Fairchild                                              */
/*       Date: 1989-1993-1995                                            */
/*          Munsell Color Science Laboratory (MCSL)                      */
/*          Center for Imaging Science                                   */
/*          Rochester Institute of Technology                            */
/*          Rochester, NY  14623                                         */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


#define LX 0.3897
#define LY 0.6890
```

```
#define LZ -0.0787
#define MX -0.2298
#define MY 1.1834
#define MZ 0.0464
#define SX 0.0000
#define SY 0.0000
#define SZ 1.0000

#define XL 1.9102
#define XM -1.1122
#define XS 0.2019
#define YL 0.3709
#define YM 0.6291
#define YS 0.0000
#define ZL 0.0000
#define ZM 0.0000
#define ZS 1.0000

#define R1 0.3804
#define R2 0.7076
#define R3 -0.0880
#define R4 -0.2151
#define R5 1.1653
#define R6 0.0480
#define R7 0.0000
#define R8 0.0000
#define R9 1.0000

#define LE 1.0000
#define ME 1.0000
#define SE 1.0000

#define nu 0.3333

#define SOFTCOPY 0
#define HARDCOPY 1

void RLAB_to_XYZ_Norm_Variable(cxLattice *in, cxLattice **out, double Xnr, double Ynr,
        double Znr, double Xn, double Yn, double Zn, double Yabsn,
                double factor,  int variable_sigma, int which_sigma_lr,
                int which_sigma_ar_br, double adjust_sigma,
                double adjust_sigma_ar_br)
{


  double *a;
  double *b;
  cxErrorCode err;
  int i,  j;
  double al, am, as, c;
  double Ytonu, LMSn[3], lmsE[3], pl, pm, ps, iCon, iCoff;
  double invMatrix[9], tmpinvMatrix[9], finalMatrix[9];
  int cogswitch;
  double tempX,  tempY,  tempZ, Xr,  Yr,  Zr;
  double sigma,  invSigma;
```

```
double sigma_ar_br, invSigma_ar_br;
double temp1, temp2;



        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,               /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
        fprintf(stdout, "I have made it here!\n");

        if(Xnr == NULL || Ynr == NULL || Znr == NULL || Xn == NULL || Yn == NULL
          || Zn == NULL)
        {
          return;
        }
        else
        {
          cxInWdgtDblSet("White Point X", Xnr);
          cxInWdgtDblSet("White Point Y", Ynr);
          cxInWdgtDblSet("White Point Z", Znr);
          cxInWdgtDblSet("Adapting WP X", Xn);
          cxInWdgtDblSet("Adapting WP Y", Yn);
          cxInWdgtDblSet("Adapting WP Z", Zn);
        }
        if(Yabsn == NULL)
        {
          return;
        }
        else
        {
          cxInWdgtDblSet("Adapting Luminance", Yabsn);
        }



        /*Decide whether want same sigma for all channels or variable sigmas
         * for luma/chromas
         */

        if(variable_sigma == 0)
        {

        /*Disable & hide ar/br sigma controls */
```

```
    cxInWdgtDisable("Sigma ar/br");
cxInWdgtDisable("Custom sigma ar/br");
    cxInWdgtHide("Custom sigma ar/br");


            /* Setup custom sigma dial range */

    cxInWdgtDblMinMaxSet("Custom sigma Lr/all",  1.0,  4.0);

    /* Decide which sigma to use        */

    switch(which_sigma_lr)
    {
       case 0:

       cxInWdgtDisable("Custom sigma Lr/all");
       cxInWdgtHide("Custom sigma Lr/all");

       sigma = (1.0 / 2.3);
       sigma_ar_br = (1.0 / 2.3);
       invSigma = 2.3;
       invSigma_ar_br = 2.3;
       break;

       case 1:

       cxInWdgtDisable("Custom sigma Lr/all");
       cxInWdgtHide("Custom sigma Lr/all");

       sigma = (1.0 / 2.9);
       sigma_ar_br = (1.0 / 2.9);
       invSigma = 2.9;
       invSigma_ar_br = 2.9;
       break;

       case 2:

       cxInWdgtDisable("Custom sigma Lr/all");
       cxInWdgtHide("Custom sigma Lr/all");

       sigma = (1.0 / 3.5);
       sigma_ar_br = (1.0 / 3.5);
       invSigma = 3.5;
       invSigma_ar_br = 3.5;
       break;

       case 3:

       cxInWdgtShow("Custom sigma Lr/all");
       cxInWdgtEnable("Custom sigma Lr/all");
       sigma = (1.0 / adjust_sigma);
       sigma_ar_br = (1.0 / adjust_sigma);
       invSigma = adjust_sigma;
       invSigma_ar_br = adjust_sigma;
```

```
        break;

    }
}

else if(variable_sigma == 1)
{
    /*Enable and show ar/br sigma controls */

    cxInWdgtEnable("Sigma ar/br");


    /* Setup custom sigma dial range */

    cxInWdgtDblMinMaxSet("Custom sigma Lr/all",  1.0,  4.0);
    cxInWdgtDblMinMaxSet("Custom sigma ar/br",  1.0,  4.0);

    /* Decide which Lr sigma to use     */

    switch(which_sigma_lr)
    {
      case 0:

      cxInWdgtDisable("Custom sigma Lr/all");
      cxInWdgtHide("Custom sigma Lr/all");

      sigma = (1.0 / 2.3);
      invSigma = 2.3;
      break;

      case 1:

      cxInWdgtDisable("Custom sigma Lr/all");
      cxInWdgtHide("Custom sigma Lr/all");

      sigma = (1.0 / 2.9);
      invSigma = 2.9;
      break;

      case 2:

      cxInWdgtDisable("Custom sigma Lr/all");
      cxInWdgtHide("Custom sigma Lr/all");

      sigma = (1.0 / 3.5);
      invSigma = 3.5;
      break;

      case 3:

      cxInWdgtShow("Custom sigma Lr/all");
      cxInWdgtEnable("Custom sigma Lr/all");
      sigma = (1.0 / adjust_sigma);
      invSigma = adjust_sigma;
      break;
```

```
        }

        switch(which_sigma_ar_br)
        {
          case 0:

          cxInWdgtDisable("Custom sigma ar/br");
          cxInWdgtHide("Custom sigma ar/br");

          sigma_ar_br = (1.0 / 2.3);
          invSigma_ar_br = 2.3;
          break;

          case 1:

          cxInWdgtDisable("Custom sigma ar/br");
          cxInWdgtHide("Custom sigma ar/br");

          sigma_ar_br = (1.0 / 2.9);
          invSigma_ar_br = 2.9;
          break;

          case 2:

          cxInWdgtDisable("Custom sigma ar/br");
          cxInWdgtHide("Custom sigma ar/br");

          sigma_ar_br = (1.0 / 3.5);
          invSigma_ar_br = 3.5;
          break;

          case 3:

          cxInWdgtShow("Custom sigma ar/br");
          cxInWdgtEnable("Custom sigma ar/br");
          sigma_ar_br = (1.0 / adjust_sigma_ar_br);
          invSigma_ar_br = adjust_sigma_ar_br;
          break;

        }

    }


/* convert XYZ to LMS */
  LMSn[0] = LX * (double) Xn + LY * (double) Yn + LZ * (double) Zn;
  LMSn[1] = MX * (double) Xn + MY * (double) Yn + MZ * (double) Zn;
  LMSn[2] = SX * (double) Xn + SY * (double) Yn + SZ * (double) Zn;

/* calculate relative chromaticities */
  lmsE[0] = (3.0 * (LMSn[0]/LE))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
  lmsE[1] = (3.0 * (LMSn[1]/ME))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
  lmsE[2] = (3.0 * (LMSn[2]/SE))/((LMSn[0]/LE) + (LMSn[1]/ME) + (LMSn[2]/SE));
```

```
/* calculate pl, pm, ps */
  Ytonu = pow((double) Yabsn,nu);

  pl = (1.0 + Ytonu + lmsE[0])/(1.0 + Ytonu + (1.0/lmsE[0]));
  pm = (1.0 + Ytonu + lmsE[1])/(1.0 + Ytonu + (1.0/lmsE[1]));
  ps = (1.0 + Ytonu + lmsE[2])/(1.0 + Ytonu + (1.0/lmsE[2]));

/* apply discounting factor to pl, pm, ps */

  pl = pl + (double) factor *(1.0-pl);
  pm = pm + (double) factor *(1.0-pm);
  ps = ps + (double) factor *(1.0-ps);

/* calculate al, am, as */
  al = pl/LMSn[0];
  am = pm/LMSn[1];
  as = ps/LMSn[2];

/* calculate c */
/* since c is no longer used, it is set to zero */
  c = 0.0;

  iCon = 1.0;
  iCoff = 0.0;


/* calculate invAinvC matrix */
  tmpinvMatrix[0] = (1.0/al) * iCon;
  tmpinvMatrix[1] = (1.0/al) * iCoff;
  tmpinvMatrix[2] = (1.0/al) * iCoff;
  tmpinvMatrix[3] = (1.0/am) * iCoff;
  tmpinvMatrix[4] = (1.0/am) * iCon;
  tmpinvMatrix[5] = (1.0/am) * iCoff;
  tmpinvMatrix[6] = (1.0/as) * iCoff;
  tmpinvMatrix[7] = (1.0/as) * iCoff;
  tmpinvMatrix[8] = (1.0/as) * iCon;

/* calculate invMinvAinvC (invMatrix) matrix */
  invMatrix[0] = XL * tmpinvMatrix[0] + XM * tmpinvMatrix[3] + XS * tmpinvMatrix[6];
  invMatrix[1] = XL * tmpinvMatrix[1] + XM * tmpinvMatrix[4] + XS * tmpinvMatrix[7];
  invMatrix[2] = XL * tmpinvMatrix[2] + XM * tmpinvMatrix[5] + XS * tmpinvMatrix[8];
  invMatrix[3] = YL * tmpinvMatrix[0] + YM * tmpinvMatrix[3] + YS * tmpinvMatrix[6];
  invMatrix[4] = YL * tmpinvMatrix[1] + YM * tmpinvMatrix[4] + YS * tmpinvMatrix[7];
  invMatrix[5] = YL * tmpinvMatrix[2] + YM * tmpinvMatrix[5] + YS * tmpinvMatrix[8];
  invMatrix[6] = ZL * tmpinvMatrix[0] + ZM * tmpinvMatrix[3] + ZS * tmpinvMatrix[6];
  invMatrix[7] = ZL * tmpinvMatrix[1] + ZM * tmpinvMatrix[4] + ZS * tmpinvMatrix[7];
  invMatrix[8] = ZL * tmpinvMatrix[2] + ZM * tmpinvMatrix[5] + ZS * tmpinvMatrix[8];


/* calculate RAM matrix   (finalMatrix)  */



  finalMatrix[0] = ((R1 * invMatrix[0]) + (R2 * invMatrix[3]) + (R3 * invMatrix[6]));
  finalMatrix[1] = ((R1 * invMatrix[1]) + (R2 * invMatrix[4]) + (R3 * invMatrix[7]));
```

```
finalMatrix[2] = ((R1 * invMatrix[2]) + (R2 * invMatrix[5]) + (R3 * invMatrix[8]));
finalMatrix[3] = ((R4 * invMatrix[0]) + (R5 * invMatrix[3]) + (R6 * invMatrix[6]));
finalMatrix[4] = ((R4 * invMatrix[1]) + (R5 * invMatrix[4]) + (R6 * invMatrix[7]));
finalMatrix[5] = ((R4 * invMatrix[2]) + (R5 * invMatrix[5]) + (R6 * invMatrix[8]));
finalMatrix[6] = ((R7 * invMatrix[0]) + (R8 * invMatrix[3]) + (R9 * invMatrix[6]));
finalMatrix[7] = ((R7 * invMatrix[1]) + (R8 * invMatrix[4]) + (R9 * invMatrix[7]));
finalMatrix[8] = ((R7 * invMatrix[2]) + (R8 * invMatrix[5]) + (R9 * invMatrix[8]));

for(i = 0; i < 9; i++)
{

fprintf(stdout, "The AM^-1 matrix value at position %d is %14.7lf\n", i, invMatrix[i]);

 }

for(i = 0; i < 9; i++)
{

fprintf(stdout, "The RAM^-1 matrix value at position %d is %14.7lf\n", i, finalMatrix[i]);

}



        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {

                Yr = pow((a[0] / 100.0), invSigma);

                temp1 = pow(Yr, sigma_ar_br);
                temp2 = a[1] / 430.0;


                Xr = pow((temp2 + temp1), invSigma_ar_br);
                Zr = pow((pow(Yr, sigma_ar_br) - (a[2] / 170.0)), invSigma_ar_br);



                tempX = ((finalMatrix[0] * Xr) + (finalMatrix[1] * Yr) +
                     (finalMatrix[2] * Zr));

                tempY = ((finalMatrix[3] * Xr) + (finalMatrix[4] * Yr) +
                     (finalMatrix[5] * Zr));

                tempZ = ((finalMatrix[6] * Xr) + (finalMatrix[7] * Yr) +
                     (finalMatrix[8] * Zr));



                b[0] = tempX / Xnr;

                b[1] = tempY / Ynr;
```

```
                b[2] = tempZ / Znr;


                a += 3;
                b += 3;

                }

        }


}
```

8.3 Color space transformations

XYZ2CIELab.c *XYZ_Norm_to_CIELab*

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


/*

        XYZ2CIELab.c

        This user function reads a lattice in and calculates CIELAB coordinates

        b[0] = L*
        b[1] = a*
        b[2] = b*


        Christopher Hauf
        April 14, 1995

*/

void XYZ_Norm_to_Lab(cxLattice *in, cxLattice **out)
{

        double *a;
        double *b;
        int i,j;
        cxErrorCode err;


        /* create the new lattice */
```

```
*out = cxLatDataNew(
        2,                  /* number of dimensions */
        in->dims,           /* dimensions array */
        in->data->nDataVar, /* number of data variables */
        cx_prim_double);/* data type */

/* use the same coordinate space as the input lattice */
cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

/* extract the data pointers */
cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

for ( i = 0; i < in->dims[1]; i++ )
{
        for ( j = 0; j < in->dims[0]; j++ )
        {

            if(a[1] > 0.008856)
            {
                b[0] = (116.0 * pow(a[1], (1.0 / 3.0)) - 16.0);
            }
            else
            {
              b[0] = (903.3 * a[1]);

            }

            if(a[0] > 0.008856 && a[1] > 0.008856)
            {
                b[1] = (500.0 * (pow(a[0], (1.0 / 3.0)) - pow(a[1], (1.0 / 3.0))));
            }
            else if(a[0] <= 0.008856 && a[1] > 0.008856)
            {
              b[1] = (500.0 * (((7.787 * a[0]) + (16.0 /116.0)) -
                    (pow(a[1], (1.0 / 3.0)))));
            }
            else if(a[0] > 0.008856 && a[1] <= 0.008856)
            {
              b[1] = (500.0 * ((pow(a[0], (1.0 / 3.0))) -
                    ((7.787 * a[1]) + (16.0 /116.0))));
            }
            else
            {
                b[1] = (500.0 * (((7.787 * a[0]) + (16.0 /116.0)) -
                    ((7.787 * a[1]) + (16.0 /116.0))));
            }




            if(a[1] > 0.008856 && a[2] > 0.008856)
            {
                b[2] = (200.0 * (pow(a[1],  (1.0 / 3.0)) - pow(a[2],  (1.0 / 3.0))));
            }
```

```
                    else if(a[1] <= 0.008856 && a[2] > 0.008856)
                    {
                     b[2] = (200.0 * (((7.787 * a[1]) + (16.0 /116.0)) -
                              (pow(a[2], (1.0 / 3.0)))));
                    }
                    else if(a[1] > 0.008856 && a[2] <= 0.008856)
                    {
                      b[2] = (200.0 * ((pow(a[1], (1.0 / 3.0))) -
                              ((7.787 * a[2]) + (16.0 /116.0))));
                    }
                    else
                    {
                     b[2] = (200.0 * (((7.787 * a[1]) + (16.0 / 116.0)) -
                              ((7.787 * a[2]) + (16.0 / 116.0))));

                    }

                        a += 3;
                        b += 3;

                }
        }

}
```

CIELab2XYZ.c *CIELab_to_XYZ_Norm*

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


/*

        CIELab2XYZ.c

        This user function reads a lattice in and calculates XYZ coordinates
        from CIELAB coordinates.  Output is X/Xn,  Y/Yn,  Z/Zn.

        b[0] = X/Xn
        b[1] = Y/Yn
        b[2] = Z/Zn


        Christopher Hauf
        April 19, 1995

*/

void Lab_to_XYZ_Norm(cxLattice *in, cxLattice **out)
{
```

```
double *a;
double *b;
double temp, fX, fY, fZ;
int i,j;
cxErrorCode err;


/* create the new lattice */
*out = cxLatDataNew(
        2,                  /* number of dimensions */
        in->dims,           /* dimensions array */
        in->data->nDataVar, /* number of data variables */
        cx_prim_double);/* data type */

/* use the same coordinate space as the input lattice */
cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

/* extract the data pointers */
cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

for ( i = 0; i < in->dims[1]; i++ )
{
        for ( j = 0; j < in->dims[0]; j++ )
        {

        /* Calculate Y/Yn   */

            if(a[0] > 8.00)
            {
                b[1] = pow(((a[0]+16.0)/116.0),  3.0);
            }
            else
            {
              b[1] = (a[0] / (116.0 * 7.787));

            }

        /* Calculate X/Xn & Z/Zn  */

            if(b[1] > 0.008856)
            {
                fX = ((a[1] / 500.0) + pow(b[1],  (1.0 / 3.0)));
                fZ = (pow(b[1],  (1.0 / 3.0)) - (a[2] / 200.0));

            }
            else
            {
                temp = ((7.787 * b[1]) + (16.0 / 116.0));

                fX = ((a[1] / 500.0) + temp);
                fZ = (temp - (a[2] /200.0));

            }
```

```
                    if(fX > pow(0.008856,  (1.0 / 3.0)))
                    {
                            b[0] = pow(fX,  3.0);
                    }
                    else
                    {
                            b[0] = ((fX - (16.0 / 116.0))/ 7.787);
                    }

                    if(fZ > pow(0.008856,  (1.0 / 3.0)))
                    {
                            b[2] = pow(fZ,  3.0);
                    }
                    else
                    {
                            b[2] = ((fZ - (16.0 / 116.0))/ 7.787);
                    }

                    if(b[0]  < 0.0)
                    {
                     b[0] = 0.0;
                    }

                    if(b[1] < 0.0)
                    {
                     b[1] = 0.0;

                    }

                    if(b[2] < 0.0)
                    {
                     b[2] = 0.0;
                    }


                        a += 3;
                        b += 3;

                }
        }

}
```

CIELab2CIELCh.c *CIELab_to_CIELCh*

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>

#define PI 3.141592653589793
```

```
/*

        CIELab2CIELCh.c

        This user function reads a lattice in and calculates CIE LCh coordinates
        from CIELAB coordinates.

        b[0] = L*
        b[1] = C*
        b[2] = hue angle


        Christopher Hauf
        April 21, 1995

*/

void Lab_to_CIELCh(cxLattice *in, cxLattice **out)
{

        double *a;
        double *b;
        double temp;
        int i,j;
        cxErrorCode err;


        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,               /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {

                b[0] = a[0];

                b[1] = sqrt(pow(a[1],  2.0) + pow(a[2],  2.0));


            temp = atan2(a[2], a[1]);
```

```
            if(temp < 0.0)
            {
                    b[2] = (360.0 + (temp * (180.0 / PI)));
            }
            else
            {
                    b[2] = (temp * (180.0 / PI));
            }


                    a += 3;
                    b += 3;

                    }
        }

}
```

CIELCh2CIELab.c ***CIELCh_to_CIELab***

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>

#define PI 3.141592653589793

/*

        CIELCh2CIELab.c

        This user function reads a lattice in and calculates CIE Lab coordinates
        from CIE LCh coordinates.

        b[0] = L*
        b[1] = a*
        b[2] = b*


        Christopher Hauf
        May 11, 1995

*/

void CIELCh_to_CIELab(cxLattice *in, cxLattice **out)
{

        double *a;
        double *b;
        double temp;
        int i,j;
        cxErrorCode err;
```

```
        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,               /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {

                b[0] = a[0];


                b[1] = (a[1] * cos((a[2] * (PI / 180.0))));

                b[2] = (a[1] * sin((a[2] * (PI / 180.0))));




                a += 3;
                b += 3;

                }
        }

}
```

XYZ2CIELuv.c *XYZ_Norm_to_CIELuv*

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


/*

        XYZ2CIELab.c
```

```
        This user function reads a lattice in and calculates CIELuv coordinates

        b[0] = L*
        b[1] = u*
        b[2] = v*


        Christopher Hauf
        February 11, 1996

*/

void XYZ_Norm_to_Luv(cxLattice *in, cxLattice **out, double whitePointX,
            double whitePointY, double whitePointZ,
                    cxParameter **uPrimeN, cxParameter **vPrimeN)
{

        double *a;
        double *b;
        double uPrime, vPrime, uPn, vPn;
        int i,j;
        cxErrorCode err;


        if(whitePointX == NULL || whitePointY == NULL || whitePointZ == NULL)
        {
          return;
        }

        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,                       /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

        /* Calculate u' and v' for the reference white */

        uPn = (4.0 * whitePointX) / (whitePointX + (15.0 * whitePointY)
                + (3.0 * whitePointZ));

        vPn = (9.0 * whitePointY) / (whitePointX + (15.0 * whitePointY)
                        + (3.0 * whitePointZ));

/*      cxParamTypeSet(uPrimeN, cx_param_double);
        cxParamTypeSet(vPrimeN, cx_param_double);

        cxParamDblSet(uPrimeN, uPn);
```

```
            cxParamDblSet(vPrimeN,  vPn);
    */

        *uPrimeN = cxParamDoubleNew(uPn);
            *vPrimeN = cxParamDoubleNew(vPn);



            for ( i = 0; i < in->dims[1]; i++ )
            {
                    for ( j = 0; j < in->dims[0]; j++ )
                    {

                        if(a[1] > 0.008856)
                        {
                              b[0] = (116.0 * pow(a[1], (1.0 / 3.0)) - 16.0);
                        }
                        else
                        {
                         b[0] = (903.3 * a[1]);

                        }

                        uPrime = (4.0 * (a[0] * whitePointX))/((a[0] * whitePointX)
                                + (15.0 * (a[1] * whitePointY))
                                        + (3.0 * (a[2] * whitePointZ)));

                        vPrime = (9.0 * (a[1] * whitePointX))/((a[0] * whitePointX)
                                + (15.0 * (a[1] * whitePointY))
                                        + (3.0 * (a[2] * whitePointZ)));



                        b[1] = (13.0 * b[0]) * (uPrime - uPn);
                        b[2] = (13.0 * b[0]) * (vPrime - vPn);



                            a += 3;
                            b += 3;

                    }
            }

}
```

sRGB2xyznorm.c ***sRGB_to_Norm_XYZ***

```
#include <stdio.h>
#include <math.h>

#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
```

```
#include <cx/cxLattice.api.h>

/*

        This function calculates normalized XYZ tristimulus values from
    input NIFRGB code values.

        Based on final Reference Monitor characteristic (6/19/96) NIF-44

        Chris Hauf
        6/20/96

*/

void NIFRGB_to_XYZ_Norm(cxLattice *in,cxLattice **out, double whitePointX,
                        double whitePointY, double whitePointZ)
{
        int     i,j,k;              /* loop variables */
        double  *xyzout;
        unsigned char *rgbin;               /* data pointers */
        double temp1, temp2, temp3, norm1, norm2, norm3;
        double val1, val2, val3;
        cxErrorCode err;


        /* create the new lattice */
        *out = cxLatDataNew(
                2,              /* number of dimensions */
                in->dims,               /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */


        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&rgbin,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&xyzout,NULL,NULL);


        if(whitePointX == NULL || whitePointY == NULL || whitePointZ == NULL)
        {
          return;
        }
        else
        {
          cxInWdgtDblSet("White Point X", whitePointX);
          cxInWdgtDblSet("White Point Y", whitePointY);
          cxInWdgtDblSet("White Point Z", whitePointZ);

        }


       /* loop over the data elements */
```

```
for ( i = 0; i < in->dims[1]; i++ )
{
        for ( j = 0; j < in->dims[0]; j++ )
        {

        norm1 = (double) rgbin[0] / 255.0;
        norm2 = (double) rgbin[1] / 255.0;
        norm3 = (double) rgbin[2] / 255.0;



        if(norm1 >= 0.00304)
        {

          temp1 = pow(((norm1 + 0.055) / 1.055),  2.4);
        }
        else
        {
           temp1 = norm1/12.92;
        }

        if(norm2 >= 0.00304)
        {

          temp2 = pow(((norm2 + 0.055)/ 1.055),  2.4);
        }
        else
        {
           temp2 = norm2/12.92;
        }

        if(norm3 >= 0.00304)
        {

          temp3 = pow(((norm3 + 0.055) /1.055),  2.4);
        }
        else
        {
           temp3 = norm3/12.92;
        }



        val1 = ((0.4124 * temp1) + (0.3576 * temp2) +
                (0.1805 * temp3));
        xyzout[0] = (norm1 / whitePointX);

        val2 = ((0.2126 * temp1) + (0.7152 * temp2) +
                (0.0722 * temp3));
        xyzout[1] = (norm2 / whitePointY);

        val3 = ((0.0193 * temp1) + (0.1192 * temp2) +
                (0.9505 * temp3));
        xyzout[2] = (norm3 / whitePointZ);
```

```
                rgbin += 3;
                xyzout += 3;
                }
        }

}
```

YCC2XYZNorm.c ***PhotoYCC_to_XYZ_Norm***

```
/*

   YCC2PCSXYZ.c

   This file reads in a lattice of YCC data and converts to PCS XYZ values

   pcsout[0] = X;
   pcsout[1] = Y;
   pcsout[2] = Z;


*/



#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>



void PhotoYCC_to_XYZNorm(cxLattice *in, cxLattice **out, double wpX, double wpY, double wpZ)
{


unsigned char     *ycc;
double *xyzout;
double l, c1, c2;
double rtemp, gtemp, btemp;
double red,  green,  blue;
double rref,  gref, bref;
double rlog, glog, blog;
double rlogp, glogp, blogp;
double Xd65,  Yd65,  Zd65;
cxErrorCode err;
int       i,  j,  k;
```

```
        /* create the new lattices */
        *out = cxLatDataNew(
                2,                  /* number of dimensions */
                in->dims,                   /* dimensions array */
                in->data->nDataVar,         /* number of data variables */
                cx_prim_double);/* data type */


        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);



        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&ycc,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&xyzout,NULL,NULL);




/* loop over the data elements */
  for ( i = 0; i < in->dims[1]; i++ )
  {
    for ( j = 0; j < in->dims[0]; j++ )
    {

        l = ( 1.402 / 255.0) * (double) ycc[0];
        c1 = (((double) ycc[1] - 156.0) / 111.40);
        c2 = (((double) ycc[2] - 135.64) / 135.64);

        rtemp = l + c2;
        gtemp = l + ( -0.194 * c1) + ( -0.509 * c2);
        btemp = l + c1;

        if(rtemp >= 0.081)
        {
          red = pow(((rtemp + 0.099) / 1.099),  (1.0 / 0.45));
        }
    else if(rtemp <= -0.081)
        {
          red = pow(((rtemp - 0.099)/ -1.099),  (1.0 / 0.45));
        }
        else
        {
          red = (rtemp / 4.5);
        }

    if(gtemp >= 0.081)
        {
          green = pow(((gtemp + 0.099) / 1.099),  (1.0 / 0.45));
        }
    else if(gtemp <= -0.081)
        {
          green = pow(((gtemp - 0.099)/ -1.099),  (1.0 / 0.45));
```

```
        }
        else
        {
          green = (gtemp / 4.5);
        }

        if(btemp >= 0.081)
        {
          blue = pow(((btemp + 0.099) / 1.099),  (1.0 / 0.45));
        }
    else if(btemp <= -0.081)
        {
          blue = pow(((btemp - 0.099)/ -1.099),  (1.0 / 0.45));
        }
        else
        {
          blue = (btemp / 4.5);
        }

        Xd65 = ((0.4124 * red) + (0.3576 * green) + (0.1805 * blue));
        Yd65 = ((0.2126 * red) + (0.7152 * green) + (0.0722 * blue));
        Zd65 = ((0.0193 * red) + (0.1192 * green) + (0.9505 * blue));

        xyzout[0] = Xd65/wpX;
        xyzout[1] = Yd65/wpX;
        xyzout[2] = Zd65/wpY;

  ycc += 3;
  xyzout += 3;

  }

 }


}
```

8.4 Mathematical operations

Offset_gain.c *Channel_Offset_Gain*

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>



void Offset_Gain(cxLattice *in,  cxLattice **out,
```

```
                double red_gain, double red_offset, double green_gain.
                double green_offset, double blue_gain, double blue_offset)
{


 cxErrorCode err;
 double *a;
 double *b;
 int i, j;
 int count;


        /* create the new lattice */
        *out = cxLatDataNew(
                2,                  /* number of dimensions */
                in->dims,                   /* dimensions array */
                in->data->nDataVar,         /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);

        count = cxDataRefCntGet(*a);
        fprintf(stderr, "Count = %d\n", count);


    if(red_gain == NULL)
    {
          red_gain = 1.0;
    }
   if(red_offset == NULL)
    {
          red_offset = 0.0;
    }
   if(green_gain == NULL)
    {
         green_gain = 1.0;
    }
   if(green_offset == NULL)
    {
          green_offset = 0.0;
    }
   if(blue_gain == NULL)
    {
          blue_gain = 1.0;
    }
   if(blue_offset == NULL)
    {
```

```
        blue_offset = 0.0;
    }

        cxInWdgtDblSet("Red gain", red_gain);
        cxInWdgtDblSet("Red offset", red_offset);
        cxInWdgtDblSet("Green gain", green_gain);
        cxInWdgtDblSet("Green offset", green_offset);
        cxInWdgtDblSet("Blue gain", blue_gain);
        cxInWdgtDblSet("Blue offset", blue_offset);



        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {


                b[0] = ((double) red_offset + ((double) red_gain * a[0]));

                b[1] = ((double) green_offset + ((double) green_gain * a[1]));

                b[2] = ((double) blue_offset + ((double) blue_gain * a[2])):



                a += 3;
                b += 3;

                }
        }




/*      cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
    cxDataRefDec(*b);
*/
}
```

power.c ***Channel_Power***

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>
```

```
void Power(cxLattice *in,  cxLattice **out,
                    double red_power, double green_power, double blue_power)

{



 cxErrorCode err;
 double *a;
 double *b;
 int i, j;



        /* create the new lattice */
        *out = cxLatDataNew(
                2,                  /* number of dimensions */
                in->dims,                   /* dimensions array */
                in->data->nDataVar,         /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);



   if(red_power == NULL)
    {
          red_power = 1.0;
    }
   if(green_power == NULL)
    {
         green_power = 1.0;
    }
   if(blue_power == NULL)
    {
          blue_power = 1.0;
    }

         cxInWdgtDblSet("Channel 1 Exponent Value", red_power);
         cxInWdgtDblSet("Channel 2 Exponent Value", green_power);
         cxInWdgtDblSet("Channel 3 Exponent Value", blue_power);


        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {
```

```
                b[0] = pow(a[0], red_power);

                b[1] = pow(a[1], green_power);

                b[2] = pow(a[2], blue_power);



                a += 3;
                b += 3;

                }
        }




/*      cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);
    cxDataRefDec(*b);
*/
}
```

sigmoid.c ***Sigmoid***

```
/*

  sigmoid.c

  This file reads in a lattice of double data scaled between 0.0 and 1.0
  and applies a sigmoid like function to it.  The user has control of the
  "slope" of the function.

  Christopher Hauf
  December 13, 1995

*/


#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>


extern double apply_sigmoid(double value, double c0);




void Sigmoid(cxLattice *in, cxLattice **out,
                double red_exp, double green_exp, double blue_exp)
```

```
cxErrorCode err;
double *a;
double *b;
int i, j;



        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,               /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */

        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);



 if(red_exp == NULL)
 {
       red_exp = 1.0;
 }
 if(green_exp == NULL)
 {
      green_exp = 1.0;
 }
 if(blue_exp == NULL)
 {
       blue_exp = 1.0;
 }

        cxInWdgtDblSet("Channel 1 Sigmoid Value", red_exp);
        cxInWdgtDblSet("Channel 2 Sigmoid Value", green_exp);
        cxInWdgtDblSet("Channel 3 Sigmoid Value", blue_exp);


       /* loop over the data elements */
       for ( i = 0; i < in->dims[1]; i++ )
       {
                for ( j = 0; j < in->dims[0]; j++ )
                {

                b[0] = apply_sigmoid(a[0],  red_exp);
                b[1] = apply_sigmoid(a[1],  green_exp);
```

```
                b[2] = apply_sigmoid(a[2],  blue_exp);


                a += 3;
                b += 3;

                }
        }


}
```

apply_sigmoid.c ***Sigmoid***

```
/*

  apply_sigmoid.c

  This file does the calculation of the value at a certain point along the
  sigmoid like curve.

  Christopher R. Hauf
  December 13, 1995
*/

#include <stdio.h>
#include <math.h>

double apply_sigmoid(double value,  double c0)
{
 double tmp1, tmp2,  tmp3;
 double result;
 double min = 0.0;
 double range = 1.0;


 tmp1 = (value - min) / range;

 if (tmp1 < 0.5) {
  if (tmp1 == 0.0) {
   tmp2 = 0.0;
  }
  else {

        tmp3 = pow((2.0*tmp1), c0);
    tmp2 = 0.5 * tmp3;
  }
 }
 else {
  if (tmp1 == 0.5) {
   tmp2 = 0.5;
```

```
  }
  else {
    result = pow(((2.0*tmp1)-1.0), (1.0/c0));

    tmp2 = 0.5 * (1.0 + tmp3);
  }
 }

 result = min + (tmp2 * range);

 if (result < min)
  result = min;

 if (result > (min + range))
  result = min + range;

 return(result);
}
```

Calc_3x3_Matrix.c ***3x3_Matrix_Multiplication***

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>



void Mult_3X3_Matrix(char *filename, int which,  cxLattice *in,  cxLattice **out,
                     double mat1x1,  double mat1x2,  double mat1x3,
                     double mat2x1,  double mat2x2,  double mat2x3,
                     double mat3x1,  double mat3x2,  double mat3x3)
{



 FILE *inputFile;
 cxErrorCode err;
 double *a;
 double *b;
 cxPrimType *inputType;
 int i, j, k;



        /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in->dims,                       /* dimensions array */
                in->data->nDataVar,     /* number of data variables */
                cx_prim_double);/* data type */
```

```
        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);


        /* extract the data pointers */
        cxLatPtrGet(in,NULL,(void **)&a,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&b,NULL,NULL);



switch(which)
{

case 0:

        if(mat1x1 == NULL && mat1x2 == NULL && mat1x3 == NULL &&
          mat2x1 == NULL && mat2x2 == NULL && mat2x3 == NULL &&
          mat3x1 == NULL && mat3x2 == NULL && mat3x3 == NULL)
        {
         return;
        }
        else
        {

        cxInWdgtDblSet("m1x1", mat1x1);
        cxInWdgtDblSet("m1x2", mat1x2):
        cxInWdgtDblSet("m1x3", mat1x3);
        cxInWdgtDblSet("m2x1", mat2x1);
        cxInWdgtDblSet("m2x2", mat2x2);
        cxInWdgtDblSet("m2x3", mat2x3);
        cxInWdgtDblSet("m3x1", mat3x1);
        cxInWdgtDblSet("m3x2", mat3x2):
        cxInWdgtDblSet("m3x3", mat3x3);

        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
                for ( j = 0; j < in->dims[0]; j++ )
                {


                b[0] = (( ((double) a[0]) * (double) mat1x1) +
                        ( ((double) a[1]) *(double) mat1x2) +
                        ( ((double) a[2]) * (double) mat1x3));

                b[1] = ((((double) a[0]) * (double) mat2x1) +
                        (((double) a[1]) * (double) mat2x2) +
                        (((double) a[2]) * (double) mat2x3));

                b[2] = ((((double) a[0]) * (double) mat3x1) +
                        (((double) a[1]) * (double) mat3x2) +
                        (((double) a[2]) * (double) mat3x3));
```

```
                a += 3;
                b += 3;

                }
        }
        }

        break;

case 1:

if(filename == NULL) return;
if(filename[0] == NULL) return;


 if((inputFile = fopen(filename, "r")) == NULL)
 {
   cxModAlert("Can't open file!");
   return;
 }
 else
 {
        fprintf(stdout,  "I have made it here past the open.\n");
 }



 (void) fscanf(inputFile, "%lf  %lf  %lf", &mat1x1,
        &mat1x2, &mat1x3);
 (void) fscanf(inputFile, "%lf %lf %lf", &mat2x1,
        &mat2x2, &mat2x3);
 (void) fscanf(inputFile, "%lf %lf %lf", &mat3x1,
        &mat3x2, &mat3x3);


/*   fprintf(stdout,  "%lf %lf %lf\n", mat1x1,  mat2x2,  mat3x3);   */


 fclose(inputFile);

        cxInWdgtDblSet("m1x1", mat1x1);
        cxInWdgtDblSet("m1x2", mat1x2);
        cxInWdgtDblSet("m1x3", mat1x3);
        cxInWdgtDblSet("m2x1", mat2x1);
        cxInWdgtDblSet("m2x2", mat2x2);
        cxInWdgtDblSet("m2x3", mat2x3);
        cxInWdgtDblSet("m3x1", mat3x1);
        cxInWdgtDblSet("m3x2", mat3x2);
        cxInWdgtDblSet("m3x3", mat3x3);


        /* loop over the data elements */
        for ( i = 0; i < in->dims[1]; i++ )
        {
```

```
                    for ( j = 0; j < in->dims[0]; j++ )
                    {


                    b[0] = (( ((double) a[0]) * mat1x1) +
                            ( ((double) a[1]) * mat1x2) +
                            ( ((double) a[2]) * mat1x3));

                    b[1] = ((((double) a[0]) * mat2x1) +
                            (((double) a[1]) * mat2x2) +
                            (((double) a[2]) * mat2x3));

                    b[2] = ((((double) a[0]) * mat3x1) +
                            (((double) a[1]) * mat3x2) +
                            (((double) a[2]) * mat3x3));


/*                  fprintf(stdout,  "%lf %lf %lf\n", (double) a[0], (double) a[1],
                      (double) a[2]);
*/

                    a += 3;
                    b += 3;

                    }
          }

     break;

   }

}
```

8.5 Look-up tables

3_1D_luts_byte.c ***3_1D_LUTS_byte***

```
#include <stdio.h>
#include <math.h>

#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>

/*
        This user functions allows the user to load 3 1-D LUTs and apply
        them on the image.

        Chris Hauf
        September 21, 1995

*/
void Three_1D_luts(cxLattice *in,cxLattice **out,  char *filename1,  char *filename2,
```

```
    char *filename3, int which1, int which2, int which3)

    int       i,j;                  /* loop variables */
    unsigned char *input;
    unsigned char *output;                  /* data pointers */
    int  LUT1[256];
    int LUT2[256];
    int LUT3[256];
    cxErrorCode err;
    FILE *fileLUT1, *fileLUT2, *fileLUT3;


    /* create the new lattice */
    *out = cxLatDataNew(
              2,                    /* number of dimensions */
              in->dims,                     /* dimensions array */
              in->data->nDataVar,           /* number of data variables */
              cx_prim_byte);    /* data type */

    /* use the same coordinate space as the input lattice */
    cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

    /* extract the data pointers */
    cxLatPtrGet(in,NULL,(void **)&input,NULL,NULL);
    cxLatPtrGet(*out,NULL,(void **)&output,NULL,NULL);

    if(filename1 == NULL || filename2 == NULL || filename3 == NULL)
    {
      return;
    }
    if(filename1[0] == NULL || filename2[0] == NULL || filename3[0] == NULL)
    {
      return;
    }


    if((fileLUT1 = fopen(filename1, "r")) == NULL)
{
  cxModAlert("Can't open file for LUT1!");
  return;
}
else
{
      fprintf(stdout,  "LUT1 opened successfully.\n");
}

if((fileLUT2 = fopen(filename2, "r")) == NULL)
{
  cxModAlert("Can't open file for LUT2!");
  return;
}
else
{
      fprintf(stdout,  "LUT2 opened successfully.\n");
}
```

```
if((fileLUT3 = fopen(filename3, "r")) == NULL)
{
  cxModAlert("Can't open file for LUT3!");
  return;
}
else
{
       fprintf(stdout,  "LUT3 opened successfully.\n");
}

for(i = 0; i < 256; i++)
{

 (void) fscanf(fileLUT1, "%d", &LUT1[i]);
 (void) fscanf(fileLUT2, "%d", &LUT2[i]);
 (void) fscanf(fileLUT3, "%d", &LUT3[i]);

}

fclose(fileLUT1);
fclose(fileLUT2);
fclose(fileLUT3);



cxInWdgtStrSet("Filename LUT 1", filename1);
cxInWdgtStrSet("Filename LUT 2", filename2);
cxInWdgtStrSet("Filename LUT 3", filename3);



    /* loop over the data elements */
    for ( i = 0; i < in->dims[1]; i++ )
    {
              for ( j = 0; j < in->dims[0]; j++ )
              {

     if(which1 == 0)
     {
               output[0] = input[0];
              }
              else
              {
              output[0] = (unsigned char) LUT1[input[0]];
              }

              if(which2 == 0)
              {
                 output[1] = input[1];
              }
              else
              {
                 output[1] = (unsigned char) LUT2[input[1]];
              }
              if(which3 == 0)
```

```
                {
                  output[2] = input[2];
                }
                else
                {
                  output[2] = (unsigned char) LUT3[input[2]];
                }


                input += 3;
                output += 3;


                }
        }

}
```

read_3component.c ***Read_3Component_Data***

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>

int Count_LUT_values(char *filename, int type);

void Read_XYZ(char *filename, int numElements, int choice,  cxLattice **out)
{


  FILE *fp;
  unsigned char *byteData;
  double *doubleData;
  int i,  j,  temp1,  temp2,  temp3;
  long dimensions[2];




  if(filename == NULL) return;
  if(filename[0] == NULL) return;



switch(choice)
{

case 0:

  numElements = Count_LUT_values(filename, 0);
```

```
cxInWdgtDblSet("LUT Length", (double) numElements);

dimensions[0] = 1;
dimensions[1] = numElements;

/* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                dimensions,                     /* dimensions array */
                3,          /* number of data variables */
                cx_prim_byte);    /* data type */


  /* Create new coordinate structure for lattice */
    cxLatPtrSet(*out,NULL,NULL,cxCoordDefaultNew(2, dimensions),NULL);

        /* extract the data pointers */
        cxLatPtrGet(*out,NULL,(void **)&byteData,NULL,NULL);



if((fp = fopen(filename, "r")) == NULL)
{
  fprintf(stdout, "Could not open data file!\n");
  return;

}

        for(j = 0; j < numElements; j++)
        {

          fscanf(fp, "%d%d%d", &temp1, &temp2, &temp3);


          byteData[0] = (unsigned char) temp1;
          byteData[1] = (unsigned char) temp2;
          byteData[2] = (unsigned char) temp3;

          byteData += 3;


        }



 fclose(fp);
 break;


case 1:

numElements = Count_LUT_values(filename, 1);
cxInWdgtDblSet("LUT Length", (double) numElements);
```

```
  dimensions[0] = 1;
  dimensions[1] = numElements;

  /* create the new lattice */
        *out = cxLatDataNew(
                  2,                    /* number of dimensions */
                  dimensions,                   /* dimensions array */
                  3,          /* number of data variables */
                  cx_prim_double);/* data type */


    /* Create new coordinate structure for lattice */
     cxLatPtrSet(*out,NULL,NULL,cxCoordDefaultNew(2, dimensions),NULL);

         /* extract the data pointers */
         cxLatPtrGet(*out,NULL,(void **)&doubleData,NULL,NULL);




  if((fp = fopen(filename, "r")) == NULL)
  {
    fprintf(stdout, "Could not open data file!\n");
    return;

  }

         for(j = 0; j < numElements; j++)
         {

           fscanf(fp, "%lf%lf%lf", &doubleData[0], &doubleData[1],
             &doubleData[2]);


           doubleData += 3;


         }



  fclose(fp);
  break;
}
}

int Count_LUT_values(char *filename, int type)
{

  int i;
  int count = 0;
  int final_count = 0;
  int value;
```

```
  FILE *fp;

  if((fp = fopen(filename,  "r")) == NULL)
  {
    fprintf(stdout, "Could not open lut file!\n");
    return(-1);
  }

 switch(type)
 {
  case 0:

  while( fscanf(fp, "%d",  &value) != EOF)
  {
    count++;
  }

  fprintf(stdout, "Count = %d\n",  count);

  fclose(fp);

  final_count = count / 3;

  return(final_count);
  break;

  case 1:

  while( fscanf(fp, "%lf",  &value) != EOF)
  {
    count++;
  }

  fprintf(stdout, "Count = %d\n",  count);

  fclose(fp);

  final_count = count / 3;
  return(final_count);
  break;
  }
}
```

8.6 Data management, support & analysis

deltaE.c ***Delta_E***

```
#include <stdio.h>
#include <math.h>
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>
```

```
#include <cx/cxParameter.api.h>


/*

        deltaE.c

        This user function calculates a Delta E*ab image and the average
        delta E for two images

        Christopher Hauf

        June 14, 1996

*/


void Delta_E(cxLattice *in1, cxLattice *in2, cxLattice **out, double average)
{

        double *lab1,  *lab2,  *de;
        int i, j;
        int dims[2];
        double temp1,  temp2,  temp3,  total;
        cxErrorCode err;


        if(in1->dims[0] != in2->dims[0] || in1->dims[1] != in2->dims[1])
        {
          cxModAlert("The images are of different sizes!  Delta E can not be calculated.");
          return;
        }



  /* create the new lattice */
        *out = cxLatDataNew(
                2,                      /* number of dimensions */
                in1->dims,                      /* dimensions array */
                1,          /* number of data variables */
                cx_prim_double);/* data type */


        /* use the same coordinate space as the input lattice */
        cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in1,&err),NULL);

        /* extract the data pointers */
        cxLatPtrGet(in1,NULL,(void **)&lab1,NULL,NULL);
        cxLatPtrGet(in2,NULL,(void **)&lab2,NULL,NULL);
        cxLatPtrGet(*out,NULL,(void **)&de,NULL,NULL);

        total = 0.0;

        for ( i = 0; i < in1->dims[1]; i++ )
        {
```

```
                    for ( j = 0; j < in1->dims[0]; j++ )
                    {
                    temp1 = pow((lab1[0] - lab2[0]), 2.0);
                    temp2 = pow((lab1[1] - lab2[1]), 2.0);
                    temp3 = pow((lab1[2] - lab2[2]), 2.0);


                    de[0] = sqrt(temp1+temp2+temp3);

                    total = total + de[0];

                    lab1 +=3;
                    lab2 +=3;
                    de += 1;

                    }
          }


          temp1 = (in1->dims[0] * in1->dims[1]);
          average = total / temp1;

          cxInWdgtDblSet("Average", average);
          fprintf(stdout, "Average DeltaE* = %lf\n", average);

}
```

switch_image.c ***Switch_Image***

```
#include <stdio.h>
#include <cx/DataAccess.h>
#include <cx/DataTypes.h>
#include <cx/cxLattice.api.h>

void switch_image(cxLattice *inlat1, cxLattice *inlat2,
                  cxLattice **outlat, int which)
{

  int port;
  void *in1, *in2, *in3;



  switch(which)
  {
  case 0:


  *outlat = cxLatDup(inlat1, 1, 1);
  port = cxOutputPortOpen("Output Image");
  cxOutputDataSet(port, *outlat);
  cxOutputDataFlush(port);
  break;
```

```
   case 1:

   *outlat = cxLatDup(inlat2, 1, 1);
   port = cxOutputPortOpen("Output Image");
   cxOutputDataSet(port, *outlat);
   cxOutputDataFlush(port);
   break;
   }


}
```

white.c ***Select_White_Point***

```
/*************************************************************************/
/* Function: white.c
 * Purpose: To allow for the selection of different CIE whitepoints
 * Author: Christopher Hauf
 * Date: October 8, 1995
 *
 /************************************************************************/


#include<stdio.h>
#include<cx/cxParameter.api.h>
#include <cx/DataAccess.h>

choose_white_point(long which, cxParameter *white_X, cxParameter *white_Y,
                   cxParameter *white_Z, double X, double Y, double Z)
{


  cxParamTypeSet(white_X, cx_param_double);
  cxParamTypeSet(white_Y, cx_param_double);
  cxParamTypeSet(white_Z, cx_param_double);

  if(which < 6)
  {
     cxInWdgtHide("White Point X");
     cxInWdgtHide("White Point Y");
     cxInWdgtHide("White Point Z");
  }
  else if (which == 6)
  {
   cxInWdgtShow("White Point X");
   cxInWdgtShow("White Point Y");
   cxInWdgtShow("White Point Z");

  }

  switch(which){

  case 0:  /* D65 */

          cxParamDblSet(white_X, 95.04);
```

```
        cxParamDblSet(white_Y, 100.0);
        cxParamDblSet(white_Z, 108.89);

 break;

 case 1: /* A */

        cxParamDblSet(white_X, 109.85);
        cxParamDblSet(white_Y, 100.0);
        cxParamDblSet(white_Z, 35.58);

 break;


case 2: /* D50 */

        cxParamDblSet(white_X, 96.42);
        cxParamDblSet(white_Y, 100.0);
        cxParamDblSet(white_Z, 82.49);

 break;



case 3: /* D55 */

        cxParamDblSet(white_X, 95.68);
        cxParamDblSet(white_Y, 100.0);
        cxParamDblSet(white_Z, 92.14);

 break;


case 4: /* D75 */

        cxParamDblSet(white_X, 94.96);
        cxParamDblSet(white_Y, 100.0);
        cxParamDblSet(white_Z, 122.61);

 break;


case 5: /* C */

    cxParamDblSet(white_X, 98.07);
        cxParamDblSet(white_Y, 100.0);
        cxParamDblSet(white_Z, 118.23);

 break;

 case 6: /* Set your own */

 if( X == NULL || Y == NULL || Z == NULL)
 {
 return;
```

```
  }
  else
  {

   cxParamDblSet(white_X, X);
   cxParamDblSet(white_Y, Y);
   cxParamDblSet(white_Z, Z);

   cxInWdgtDblSet("White Point X", X);
   cxInWdgtDblSet("White Point Y", Y);
   cxInWdgtDblSet("White Point Z", Z);

  }
  break;

 }


}
```